

Institut für Angewandte Informatik und  
Formale Beschreibungsverfahren (AIFB)  
Karlsruher Institut für Technologie (KIT)

**Diplomarbeit**

## **Evolution künstlicher dreidimensionaler Wesen**

Autor: Fabian Rigoll  
Matrikelnr.: 1360350

Betreuer: Dipl.-Inf. Daniel Pathmaperuma  
Dipl.-Inf. Lukas König  
Referent: Prof. Dr. Hartmut Schmeck

Abgabedatum: 31.10.2011



Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, 31. Oktober 2011

Fabian Rigoll



Diese Arbeit untersucht die Evolution künstlicher Wesen in einer dreidimensionalen Umgebung. Auf Basis eines bestehenden Frameworks wurde dazu ein Simulationswerkzeug entworfen, mit welchem die physikalisch korrekte Simulation einer dreidimensionalen Welt möglich ist. Des Weiteren wurden die zu untersuchenden Wesen – auch Agenten genannt – entwickelt und programmiert. Ein Agent besteht aus Quadern, welche durch Gelenke miteinander verbunden sind. Die Gelenke können durch ein Künstliches Neuronales Netz vom Agenten gesteuert werden. Weiterhin können dem Agenten Sensoren zugewiesen werden, die ihm Informationen über seine Umwelt liefern.

Mit der Verwendung eines Evolutionären Algorithmus war es möglich, Populationen von sich fortbewegenden Agenten zu erzeugen. Der Prozess der Evolution wurde ebenso untersucht wie die Parameter, die diesen Prozess beeinflussen. Es konnte gezeigt werden, dass selbst einfach aufgebaute Agenten so evolviert werden können, dass sie in der Lage sind, sich selbstständig in einer dreidimensionalen Welt fortzubewegen.



# Inhaltsverzeichnis

<b>1. Einleitung und Motivation</b>	<b>15</b>
<b>2. Grundlagen und Problemstellung</b>	<b>19</b>
2.1. Aufbau der Arbeit . . . . .	19
2.2. Problemstellung . . . . .	20
2.3. Grundlagen und Begriffe . . . . .	20
2.3.1. Evolution . . . . .	21
2.3.1.1. Definition . . . . .	21
2.3.1.2. Geschichte der Evolutionstheorie . . . . .	21
2.3.1.3. Die vier Grundpfeiler der Evolutionstheorie . . . . .	23
2.3.2. Evolutionäre Algorithmen . . . . .	23
2.3.2.1. Begriffsdefinitionen . . . . .	24
2.3.2.2. Ablauf Evolutionärer Algorithmen . . . . .	26
2.3.2.3. Body Brain Co-Evolution vs. Koevolution . . . . .	26
2.3.3. Künstliche Neuronale Netze . . . . .	27
2.3.3.1. Allgemeines zu Künstlichen Neuronalen Netzen . . . . .	27
2.3.3.2. Feed-Forward-Netze und rekurrente Netze . . . . .	33
2.3.3.3. Evolution statt Lernverfahren . . . . .	34
2.4. Verwandte Arbeiten . . . . .	34
2.4.1. Pionierleistungen von SIMS . . . . .	35
2.4.2. Evolutionary Robotics von NOLFI und FLOREANO . . . . .	36
2.4.3. Erweiterung der Arbeiten von SIMS durch KRČAH . . . . .	37

2.4.4.	Arbeiten von LEHMANN und STANLEY . . . . .	38
2.4.5.	Erweiterung der Arbeiten von SIMS durch LASSABE ET AL. . . . .	38
2.4.6.	Künstliche Neuronale Netze nach FAHLMANN . . . . .	38
2.4.7.	Arbeiten von HELAOUI . . . . .	39
2.4.8.	Arbeiten über Künstliche Neuronale Netze von MÜL- LER, NAGEL UND COLLING . . . . .	39
<b>3.</b>	<b>Entwurf und Implementierung</b>	<b>41</b>
3.1.	Erstellung einer Simulationsumgebung . . . . .	41
3.1.1.	Das EAS-Framework . . . . .	41
3.1.2.	Auswahl einer 3D-Physik-Engine . . . . .	43
3.1.2.1.	Anforderungen an eine 3D-Physik-Engine . . . . .	43
3.1.2.2.	JBULLET als 3D-Physik-Engine . . . . .	45
3.1.2.3.	Funktionsweise von JBULLET . . . . .	49
3.1.2.4.	Vereinfachungen . . . . .	56
3.1.2.5.	Ein Video-Plugin zur grafischen Darstellung . . . . .	58
3.2.	Entwurf und Umsetzung . . . . .	60
3.2.1.	Der Agent . . . . .	60
3.2.1.1.	Prototyp eines dreidimensionalen Agenten . . . . .	60
3.2.1.2.	Umsetzung des Prototyps . . . . .	62
3.2.2.	Das Genom . . . . .	68
3.2.2.1.	Grundlegende Eigenschaften . . . . .	68
3.2.2.2.	Aufbau des Genoms . . . . .	69
3.2.2.3.	Einordnung des Genoms . . . . .	72
3.2.3.	Fortpflanzung . . . . .	72
3.2.3.1.	Mutationen . . . . .	73
3.2.3.2.	Crossover . . . . .	77
3.3.	Implementierung eigener Klassen . . . . .	79



<b>4. Experimente</b>	<b>89</b>
4.1. Entwurf der Experimente . . . . .	89
4.1.1. Probeläufe und Entwicklung der Experimente . . . .	89
4.1.1.1. Testphase und Auswahl der geeigneten Gelenke . . . . .	90
4.1.1.2. Entwicklung erster Agenten . . . . .	90
4.1.1.3. Einbeziehung Künstlicher Neuronaler Netze	91
4.1.1.4. Lösung von Problemen bei der Simulation .	91
4.1.2. Grundsätzlicher Ablauf eines Experiments . . . . .	93
4.1.3. Erklärung der Parameter . . . . .	96
4.2. Durchführung und Auswertung . . . . .	100
4.2.1. Abschließende Versuchsreihen . . . . .	100
4.2.1.1. Verwendete Parameter . . . . .	100
4.2.1.2. Anmerkungen zu den gewählten Parametern	103
4.2.1.3. Überblick über die Experimente . . . . .	104
4.2.2. Erste Versuchsreihe mit normaler Fitnessfunktion .	106
4.2.2.1. Ergebnisse der ersten Versuchsreihe mit normaler Fitnessfunktion . . . . .	106
4.2.2.2. Auswertung der Ergebnisse aus der ersten Versuchsreihe . . . . .	107
4.2.2.3. Schlussfolgerung aus den Ergebnissen der ersten Versuchsreihe . . . . .	112
4.2.3. Zweite Versuchsreihe mit modifizierter Fitnessfunktion	114
4.2.3.1. Ergebnisse der zweiten Versuchsreihe mit modifizierter Fitnessfunktion . . . . .	115
4.2.3.2. Auswertung der Ergebnisse aus der zweiten Versuchsreihe . . . . .	116
4.2.3.3. Schlussfolgerung aus den Ergebnissen der zweiten Versuchsreihe . . . . .	120
4.3. Beurteilung der Ergebnisse . . . . .	121
4.4. Anmerkung zur Robustheit der Ergebnisse . . . . .	122

<b>5. Zusammenfassung und Ausblick</b>	<b>127</b>
5.1. Zusammenfassung der Arbeit . . . . .	127
5.2. Ausblick . . . . .	129
5.2.1. Weitere Forschungsmöglichkeiten . . . . .	129
5.2.2. Anwendung der Ergebnisse . . . . .	131
<b>A. Vorstellung einiger Simulationsläufe</b>	<b>133</b>
A.1. Erste Versuchsreihe mit normaler Fitnessfunktion . . . . .	133
A.1.1. Verlauf der Fitness . . . . .	133
A.1.2. Pfade . . . . .	135
A.2. Zweite Versuchsreihe mit modifizierter Fitnessfunktion . . . . .	137
A.2.1. Verlauf der Fitness . . . . .	137
A.2.2. Pfade . . . . .	139
<b>B. DVD</b>	<b>143</b>

# Abbildungsverzeichnis

2.1. Schematische Darstellung eines Neurons und das Modell von McCULLOCH und PITTS . . . . .	29
2.2. Künstliches Neuronales Netz . . . . .	31
2.3. Matrixdarstellung eines Künstlichen Neuronalen Netzes . . . . .	32
3.1. Rechtshändiges Koordinatensystem von JBULLET . . . . .	53
3.2. Zufällig erzeugter Agent in der dreidimensionalen Simulationsumgebung . . . . .	61
4.1. Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment <i>D1</i> . . . . .	108
4.2. Zurückgelegter Pfad des besten Agenten aus Lauf 6 des Experiments <i>D1</i> . . . . .	111
4.3. Zurückgelegter Pfad des besten Agenten aus Lauf 3 des Experiments <i>D1</i> . . . . .	112
4.4. Fitnessverlauf von Simulation 3 aus Experiments <i>D1</i> . . . . .	113
A.1. Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment <i>L1</i> . . . . .	134
A.2. Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment <i>U1</i> . . . . .	134
A.3. Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment <i>V1</i> . . . . .	135

A.4. Zurückgelegte Pfade eines fehlerhaften und eines normalen Agenten des Experiments <i>L1</i> . . . . .	136
A.5. Zurückgelegte Pfade eines fehlerhaften und eines normalen Agenten des Experiments <i>U1</i> . . . . .	136
A.6. Zurückgelegte Pfade eines fehlerhaften und eines normalen Agenten des Experiments <i>V1</i> . . . . .	137
A.7. Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment <i>S2</i> . . . . .	138
A.8. Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment <i>W2</i> . . . . .	138
A.9. Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment <i>X2</i> . . . . .	139
A.10. Zurückgelegte Pfade von zwei ausgewählten Agenten des Experiments <i>S2</i> . . . . .	140
A.11. Zurückgelegte Pfade von zwei ausgewählten Agenten des Experiments <i>W2</i> . . . . .	141
A.12. Zurückgelegte Pfade von zwei ausgewählten Agenten des Experiments <i>X2</i> . . . . .	141

# Tabellenverzeichnis

4.1. Parameter, die in den durchgeführten Experimenten verwendet wurden . . . . .	101
4.2. Überblick über die durchgeführten Experimente . . . . .	105
4.3. Erste Versuchsreihe: Maximal erreichte Fitness mit normaler Fitnessfunktion . . . . .	106
4.4. Maximal erreichte Fitnesswerte der einzelnen Simulationsläufe des Experiments <i>D1</i> . . . . .	110
4.5. Zweite Versuchsreihe: Maximal erreichte Fitness mit modifizierter Fitnessfunktion . . . . .	115
4.6. Vergleich der durchschnittlich erreichten maximalen Fitnesswerte bei unterschiedlichen Populationsgrößen . . . . .	125
4.7. Vergleich der durchschnittlich erreichten maximalen Fitnesswerte zwischen Feed-Forward-Netz und rekurrentem Netz . . . . .	125
4.8. Vergleich der durchschnittlich erreichten maximalen Fitnesswerte zwischen Agenten mit und Agenten ohne Lagesensor . . . . .	125
4.9. Vergleich der durchschnittlich erreichten maximalen Fitnesswerte zwischen Agenten mit und Agenten ohne Bewegungssensor . . . . .	126



# 1. Einleitung und Motivation

Betrachtet, forschet,  
die Einzelheiten sammelt,  
Naturgeheimnis werde  
nachgestammelt.

---

*Johann Wolfgang  
von Goethe*

Unser Planet ist bevölkert von einer Vielzahl von verschiedenen Lebewesen. Vom Pantoffeltierchen bis hin zum Blauwal, von der Weinbergschnecke zum Geparden, von der Eintagsfliege zum Menschen. Gemäß der Evolutionstheorie haben sich alle Lebewesen über die Jahrmlionen aus einer Ursuppe entwickelt und an ihre Umwelt angepasst. Die Natur hat so unterschiedlichste an ihre Umgebung adaptierte Erdenbewohner hervorgebracht.

Neben einigen anderen Lebewesen hat sich vor allem der Mensch im Laufe der Zeit Werkzeuge zu Nutze gemacht. Den Anfang bildeten Steine und Stöcke, später waren es Metallwerkzeuge und mechanische Geräte. Heute sind es hochkomplexe Maschinen wie Computer, Raumfähren oder auch Magnetresonanztomographen. Die Herstellung eines einfachen Speers als Werkzeug nimmt für eine einzelne Person höchstens ein paar Tage in Anspruch. Alleine der Entwurf und die Entwicklung einer hochkomplexen Maschine kann jedoch Jahre oder gar Jahrzehnte dauern.

Evolutionäre Algorithmen versuchen, den Entwicklungsprozess der Evolution nachzuahmen und zur Problemlösung verwendbar zu machen. Wäh-

rend die Evolution in der Natur bereits viele Millionen Jahre voranschreitet, ist die Laufzeit eines Evolutionären Algorithmus mit einigen Minuten bis Tagen im Vergleich winzig. Dennoch ist es möglich, gewisse Probleme mithilfe dieser Algorithmen zu lösen.

Hierzu zählt möglicherweise auch die Konstruktion komplexer Maschinen. In der Regel werden solche Maschinen heutzutage in mühevoller Ingenieursarbeit entworfen, verbessert und manchmal ganz oder zu Teilen wieder verworfen. Diese Art der Konstruktion dauert oft viele Jahre und verursacht immense Kosten. Darum wird versucht, andere Wege zu finden, um die komplexen Werkzeuge des Menschen zu entwerfen. Die Nachahmung der Evolution könnte ein ebensolcher Weg sein.

In Evolutionären Algorithmen passen sich Individuen so an, dass diese die an sie gestellten Anforderungen in einer bestimmten Umgebung besser erfüllen können. Der Begriff Individuen umfasst sowohl Lösungen für mathematische Optimierungsprobleme als auch simulierte Lebewesen, Roboter, Maschinen oder Ähnliches. Die Individuen müssen im Rahmen des Evolutionären Algorithmus bewertet werden. Eine sogenannte Fitnessfunktion übernimmt die Aufgabe der Evaluation. Alleine die Bewertung der Güte der Problemlösung ist ausreichend, um die Eignung eines Individuums zu beurteilen. Es ist nicht nötig, die Art und Weise, wie ein Problem gelöst wird, zu bewerten.

Dies ist der bemerkenswerte Vorteil der Evolutionären Algorithmen. Es ist nicht mehr erforderlich, zu definieren, *wie* ein Problem gelöst werden soll. Es muss lediglich beurteilt werden, *wie gut* ein Lösungskandidat ein Problem löst. Somit könnten Maschinen entworfen werden, deren Komplexität das übersteigt, was ein Mensch selbst zu konstruieren vermag.

Die vorliegende Arbeit hat das Ziel, die Evolution einfacher künstlicher Wesen zu untersuchen. Bei diesen Wesen handelt es sich um dreidimensionale, aus Quadern aufgebaute Roboter, die über ein Künstliches Neuronales Netz gesteuert werden. Es wird versucht, mithilfe eines Evolutionären Algorithmus solche Individuen hervorzubringen, die in der Lage sind, sich selbst kontrolliert fortzubewegen. Um diese Aufgabe zu erfüllen, soll dem Beispiel der Natur gefolgt werden. Es wird nicht vorgegeben *wie* die Fortbewegung



---

erfolgen soll, sondern lediglich bewertet, *wie gut* ein Versuchsobjekt das gestellte Problem gelöst hat.

In Zukunft wird es so möglich sein, komplexeste Wesen oder Maschinen durch Evolution zu erzeugen und zu simulieren. Wenn es gelingt, diese simulierten Maschinen dann auch tatsächlich herzustellen, wäre der langwierige Weg einer Konstruktion durch den Menschen vermeidbar.

Die Natur ist dabei das große Vorbild für die Vielfalt und Vollendung von Lösungen für Probleme.



## 2. Grundlagen und Problemstellung

Ist nicht der Anfang und  
das Ende jeder  
Wissenschaft in Dunkel  
gehüllt?

---

*Heinrich von Kleist*

### 2.1. Aufbau der Arbeit

Die vorliegende Arbeit besteht aus fünf Kapiteln.

Im ersten Kapitel findet sich eine kurze Einleitung in das diese Arbeit umfassende Themengebiet. Des Weiteren wird die Motivation, dieses Themengebiet zu betrachten, erläutert.

In diesem, dem zweiten, Kapitel werden die für das Verständnis der Arbeit notwendigen Grundlagen dargelegt. Neben verschiedenen Konzepten finden sich hier auch Begriffserklärungen und Definitionen. Anschließend wird die Problemstellung der Arbeit erläutert und ein Überblick über verwandte Arbeiten gegeben.

Das dritte Kapitel stellt den Kern dieser Arbeit dar. In diesem werden zuerst der Entwurf und die Entwicklung einer dreidimensionalen Simulationsumgebung zur Untersuchung der Evolution künstlicher Wesen beschrieben. Im Anschluss werden die Implementierung und Umsetzung der zuvor angestellten Überlegungen erläutert.

Die beschriebene Simulationsumgebung kommt in verschiedenen Experimenten zum Einsatz. Diese werden im vierten Kapitel entworfen, durchgeführt und ausgewertet.

Den Abschluss der vorliegenden Arbeit stellt das fünfte Kapitel dar, in welchem eine Zusammenfassung und ein Ausblick erfolgen.

Im Anhang finden sich Abbildungen und Erklärungen zu diversen Versuchen, deren Ergebnisse auffällig oder interessant waren. Des Weiteren ist eine DVD beigelegt, welche alle Dateien und Versuchsdaten enthält, die Inhalt und Ergebnis dieser Arbeit waren.

### 2.2. Problemstellung

Das Ziel der vorliegenden Arbeit ist die Untersuchung der Evolution von künstlichen Wesen, die sich in einer dreidimensionalen Welt entwickeln.

Um die Aufgabenstellung bearbeiten zu können, müssen zu Beginn die nötigen Grundlagen betrachtet werden. Anschließend ist ein geeignetes Simulationswerkzeug auszuwählen oder zu entwerfen, mit welchem dreidimensionale, physikalisch korrekte Simulationen von künstlichen Wesen durchgeführt werden können. Diese Wesen sind mit allen Eigenschaften zu entwickeln und zu programmieren.

Die künstlichen dreidimensionalen Wesen sollen sich durch eine nachgeahmte Evolution so entwickeln, dass sie sich selbst fortbewegen können. Um diese zu simulieren, muss ein evolutionärer Algorithmus ausgewählt und implementiert werden.

Sobald die Simulationsumgebung eingerichtet und lauffähig ist, sind Experimente zu entwerfen und durchzuführen. Die aus den Versuchen hervorgegangenen Ergebnisse sollen abschließend ausgewertet und interpretiert werden.

### 2.3. Grundlagen und Begriffe

Dieser Abschnitt erläutert einige, für das Verständnis der hier behandelten Themen wichtige, Grundlagen und Begriffe. Der begrenzte Umfang dieser

Arbeit gestattet es nicht, alle Aspekte im Detail zu diskutieren. Daher finden sich zu den meisten Unterabschnitten Literaturangaben, mithilfe derer eine Vertiefung des Themas möglich ist.

### 2.3.1. Evolution

#### 2.3.1.1. Definition

Der Begriff Evolution ist auf das lateinische Wort *evolvere*, zu deutsch etwa *entfalten*, zurückzuführen. Er wird im allgemeinen Sprachgebrauch für gewöhnlich mit (schrittweiser) Veränderung gleichgesetzt. Im biologischen Sinne beschreibt Evolution die Veränderung von Eigenschaften einer Gruppe von Organismen über den Verlauf von Generationen. Die Entwicklung eines einzelnen Individuums ist keine Evolution. [Fut07]

#### 2.3.1.2. Geschichte der Evolutionstheorie

Die Evolutionstheorie wird häufig mit dem Namen DARWIN in Verbindung gebracht. Er revolutionierte mit seinem Werk *On the Origin of Species by Means of Natural Selection* [Dar59] die Sicht der Biologie auf die Entstehung der Arten. Es gab allerdings bereits vor DARWIN ähnliche Theorien, die die Entstehung verschiedener Arten zu erklären versuchten. Die wichtigste dieser Theorien wurde von JEAN-BAPTISTE PIERRE ANTOINE DE MONET, CHEVALIER DE LAMARCK im Jahr 1809 in seinem Werk *Philosophie Zoologique* [Lam09] aufgestellt.

LAMARCK vertrat die These, dass sich alle Arten aus nicht-lebender Materie bildeten. Diese Entstehung erfolge zufällig und da gewisse Arten älter seien als andere, könne man verschiedene Entwicklungsstadien beobachten. Verschiedene Einflüsse führten laut dieser Theorie dazu, dass unterschiedliche Arten unterschiedliche Formen annähmen. Als bekanntestes Beispiel nennt er die Giraffe mit ihrem langen Hals. Dieser sei entstanden, weil eine Giraffe ihr Futter in Form von Blättern in hohen Bäumen suchen müsse.

Die Eigenschaften einer Art werden gemäß der Theorie vererbt, so dass sie über Generationen erhalten bleiben. Auch wenn sich die von LAMARCK

aufgestellte Theorie als falsch herausstellte, so war er doch der erste, der eine schlüssige Evolutionstheorie präsentierte. [Fut07]

CHARLES ROBERT DARWIN sollte mit seinem 1859 veröffentlichten Buch *On the Origin of Species by Means of Natural Selection* [Dar59] den Grundstein für die heute gültige Evolutionstheorie legen. Während einer fast fünfjährigen Forschungsreise (Dezember 1831 bis Oktober 1836) entlang der Küste Südamerikas hatte er die Möglichkeit, die dortige Flora und Fauna zu beobachten und zu untersuchen. Seine angestellten Forschungen ließen ihn zu der Überzeugung gelangen, dass alle lebenden Arten gemeinsame Vorfahren haben müssen und sich durch Evolution entwickelt haben. Da er erkannte, wie weitreichend aber auch kontrovers seine revolutionären Ideen sein würden, sammelte er über zwanzig Jahre weitere Beispiele und Belege für seine Theorie, um schließlich sein Buch zu veröffentlichen. Er postulierte, dass alle lebenden Arten von einer altertümlichen Art abstammen und dass die Natürliche Selektion den Mechanismus der Evolution darstellt. [CR06, Fut07]

In den Jahrzehnten nach DARWINS Veröffentlichung wurde seine Evolutionstheorie kontrovers diskutiert. Anhänger LAMARCKS entwickelten auf seiner ursprünglichen Veröffentlichung aufbauende, neue Theorien, die der Darwinistischen Evolutionstheorie widersprachen. In den 1930er und 1940er Jahren wurden jedoch weitere Arbeiten veröffentlicht, die DARWINS Theorie stützten und belegten. Im Jahr 1959 gelang es MILLER und UREY schließlich, experimentell zu zeigen, dass sich aus einer sogenannten „Ursuppe“ unter bestimmten Bedingungen organische Verbindungen bilden können. Aus diesen hätten wiederum erste Lebewesen entstehen können, was die Evolutionstheorie weiter untermauert. [SS96]

Die aus den Arbeiten DARWINS hervorgegangene Evolutionstheorie wird heute als stichhaltige Theorie für die Entstehung und Entwicklung der Arten angesehen. [CR06, Fut07]

### 2.3.1.3. Die vier Grundpfeiler der Evolutionstheorie

FLOREANO und MATTIUSI erläutern in [FM08] die folgenden vier Grundpfeiler der Evolutionstheorie.

**Population** Die Evolution erfordert eine Population von mindestens zwei Individuen. Wenn nur ein Individuum vorhanden ist, kann nicht von Evolution gesprochen werden.

**Diversität** Die Individuen einer Population müssen sich zumindest zu einem gewissen Grad voneinander unterscheiden.

**Vererbung** Bestimmte Eigenschaften müssen von Elternteilen auf ihre Nachkommen vererbbar sein. Ohne eine Vererbung ist keine Evolution möglich.

**Selektion** Nicht allen Individuen einer Population ist es möglich, sich fortzupflanzen. Nur ein ausgewählter Teil darf seine Eigenschaften an Nachkommen weitergeben.

Des Weiteren muss erkannt werden, dass Evolution immer zu einer bestimmten Zeit in einer bestimmten Umgebung stattfindet. Insbesondere ist die Selektion der Individuen davon abhängig. Dies impliziert auch, dass die Umgebung sich ändern kann und dass Individuen, die zuvor sehr gut angepasst waren, in dieser veränderten Umgebung nicht mehr überleben können. [FM08]

### 2.3.2. Evolutionäre Algorithmen

Die natürliche Evolution, wie sie im vorherigen Unterabschnitt beschrieben wurde, kann als Vorbild für eine Klasse von Algorithmen dienen. Den Evolutionären Algorithmen ist dabei gemeinsam, „dass sie Vorgänge und Begriffe aus der Biologie entlehnen, um daraus in einem anderen Zusammenhang Verfahren zur Lösung von Optimierungsproblemen zu beschreiben.“ [Wei07] Evolutionäre Algorithmen ahmen die in der Natur beobachtete Evolution nach, um Optimierungsprobleme zu lösen. Die oftmals

große Diskrepanz zwischen den gehegten Erwartungen und den tatsächlich zu realisierenden Lösungen zeigt u.a. [Mic07] auf.

### 2.3.2.1. Begriffsdefinitionen

Wenn von Evolutionären Algorithmen die Rede ist, werden häufig Begriffe aus der Genetik verwendet. Die wichtigsten Begriffe sollen angelehnt an [FM08, GKK04, NF01, Wei07] im Folgenden kurz erläutert werden.

**Evolution** Im Sinne von Evolutionären Algorithmen bedeutet Evolution die Veränderung der Eigenschaften einer Population von Genomen. Evolution findet zu einer bestimmten Zeit in einer bestimmten Umgebung statt und wird auch von dieser beeinflusst.

**Evolutionäre Robotik** Wenn Evolutionäre Algorithmen verwendet werden, um autonome Roboter zu erstellen, so bezeichnet man dies als Evolutionäre Robotik.

**Genom** Die Gesamtheit aller gespeicherten Eigenschaften wird Genom genannt. In der Natur besteht das Genom aus der Gesamtheit der Chromosomen bzw. der DNS<sup>1</sup>. Die Chromosomen stellen quasi die chemische Speicherung der Eigenschaften eines Organismus dar. In Evolutionären Algorithmen umfassen Genome ebenfalls alle Eigenschaften, wobei die Speicherung hier als Zeichenkette, binär, durch reelle Zahlenwerte o.Ä. erfolgen kann.

**Gen** Ein Gen legt eine Teileigenschaft fest. In Chromosomen werden Gene durch einzelne Teilstücke codiert. Bei Evolutionären Algorithmen versteht man unter Genen einzelne Eigenschaften, die in ihrer Gesamtheit in einem Genom gespeichert sind. Es kann sich dabei um reelle Zahlen, ganze Zahlen, Vektoren, Zeichenketten o.Ä. handeln.

**Genotyp** Die formale Codierung einer Lösung wird Genotyp genannt.

---

<sup>1</sup>Desoxyribonukleinsäure. Das Molekül, welches in allen Lebewesen vorkommt und die Erbinformationen trägt.



**Phänotyp** Als Phänotyp wird das Erscheinungsbild eines Lebewesens bzw. im Rahmen eines Evolutionären Algorithmus das Erscheinungsbild eines Lösungskandidaten bezeichnet. Der Phänotyp stellt somit die Realisierung des Genotyps dar.

**Individuum** Das Genom in Kombination mit dem Phänotyp wird häufig als Individuum bezeichnet. In dieser Arbeit wird der Begriff Agent analog verwendet.

**Population** Eine Menge von Individuen oder eine Menge von Genomen wird als Population bezeichnet.

**Fitness** Die Qualität oder Güte einer Lösung wird in einem Evolutionären Algorithmus Fitness genannt. Eine Fitnessfunktion weist einem Individuum in Abhängigkeit seiner Güte einen Fitnesswert zu.

**Generation** Der Begriff Generation bezeichnet eine Population zu einem bestimmten Zeitpunkt.

**Reproduktion** Das Erzeugen von Nachkommen aus einem oder mehreren Lebewesen wird Reproduktion genannt. Im Rahmen von Evolutionären Algorithmen werden aus einem oder mehreren Genomen ebenfalls Nachkommen in Form von Genomen erzeugt.

**Genetische Operatoren** Mutation und Rekombination sind genetische Operatoren.

**Mutation** Von einer Mutation spricht man, wenn eines oder mehrere Gene eines Genoms verändert werden. Diese Veränderung erfolgt in der Regel zufällig.

**Rekombination** Das Erzeugen eines neuen Genoms aus zwei oder mehr Elterngenomen wird als Rekombination bezeichnet. In der Literatur ist häufig der englische Begriff *Crossover* zu finden.

### 2.3.2.2. Ablauf Evolutionärer Algorithmen

GERDES ET AL. [GKK04] zeigen die Parallele zwischen biologischer Evolution und Evolutionären Algorithmen auf.

„Die Evolution selbst ist ein ständiger Kreislauf von Neuschaffung, Überlebensprüfung, Vermehrung der Besten und Veränderung des Vorhandenen, um sich immer wieder neu anzupassen und zu verbessern.“

Analog dazu haben Evolutionäre Algorithmen in der Regel folgenden Ablauf.

1. Schaffen einer Startpopulation
2. Berechnen der Fitness
3. Selektion
4. Anwendung genetischer Operatoren

Dieser Kreislauf wird so lange wiederholt, bis ein Abbruchkriterium erreicht wird. Der erste Schritt wird dabei nur zu Beginn des Algorithmus durchgeführt.

### 2.3.2.3. Body Brain Co-Evolution vs. Koevolution

In den Genomen der in dieser Arbeit untersuchten Agenten sind sowohl Informationen über ihren Körper als auch über ihr Gehirn gespeichert. Im Laufe eines Evolutionären Algorithmus werden die Genome durch Mutationen verändert. Die Veränderungen können dabei sowohl das Gehirn als auch den Körper des Wesens betreffen. Diesen Vorgang beschreibt KRČAĤ in [KT10, Krč10] als *Body Brain Co-Evolution* und FLOREANO ET AL. bezeichnen ihn als *Coevolution of Body and Control* [FM08].

Der Begriff Koevolution ist nach [CR06, S. 1411] jedoch abweichend definiert.

„Koevolution bezieht sich auf reziproke evolutionäre Anpassung zwischen zwei Arten. Durch die Veränderung einer Art wird ein Selektionsdruck auf eine andere ausgeübt, und diese Gegenanpassung fördert wiederum die evolutionäre Abwandlung der ersten Art.“

Diese Art der gegenseitigen Anpassung ist häufig in Räuber/Beute- oder Konkurrenz-Simulationen zu finden. SIMS [Sim94a] verwendet den Begriff *Co-Evolution* in genau diesem Sinne. In [NF01] wird die nach [CR06] definierte Koevolution *Competitive Co-Evolution* genannt. Auch hier ist die Charakterisierung durch einen Wettbewerb vorhanden.

In dieser Arbeit findet keine Koevolution zweier Arten statt. Vielmehr findet die gleichzeitige Evolution von Körper und Gehirn statt. Um Unklarheiten und Verwirrungen zu vermeiden, wird hier auf die Begriffe *Co-Evolution* und Koevolution verzichtet. Stattdessen sei von gleichzeitiger oder simultaner Evolution die Rede, wenn das gemeinsame Entwickeln von Körper und Gehirn gemeint ist.

### 2.3.3. Künstliche Neuronale Netze

#### 2.3.3.1. Allgemeines zu Künstlichen Neuronalen Netzen

Das Gehirn des Menschen ist wohl das komplexeste aller bekannten Organe. Es verfügt über etwa einhundert Milliarden ( $10^{11}$ ) Nervenzellen, die Neuronen genannt werden. Ein Neuron kann wiederum bis zu  $10^4$  Verbindungen zu anderen Neuronen aufweisen. [Hak96]

Ein typisches Neuron besteht aus drei Hauptstrukturen, welche Dendritenbaum, Zellkörper und Axon genannt werden. Der Dendritenbaum bildet die Eingabe eines Neurons. Er summiert die Ausgabesignale der umgebenen Neuronen auf und leitet diese an den Zellkörper weiter. Die Signale werden in Form eines elektrischen Potentials übermittelt. Sobald das aufsummierte elektrische Potential der Eingabe einen Schwellenwert überschreitet, erzeugt der Zellkern einen elektrischen Impuls, welcher vom Axon fortgeleitet wird. [Pri11, RMS91]

Das Axon verzweigt sich und kann so den Zellkörper mit zahlreichen weiteren Neuronen verbinden. Die Kontaktstellen eines Axons, welche sich entweder auf dem Dendritenbaum oder auf dem Zellkörper direkt befinden, werden Synapsen genannt. Die meisten Synapsen wandeln die elektrischen Impulse der Axone dergestalt um, dass Überträgerstoffe ausgeschüttet werden. Es handelt sich also um chemische Kontakte, die elektrisch angesteuert werden. [RMS91] Die linke Hälfte von Abbildung 2.1 zeigt die schematische Darstellung eines Neurons. Der hochkomplexe Aufbau eines Gehirns aus einer Vielzahl von Neuronen verhilft den Menschen zu seinen intelligenten Leistungen, seinen Emotionen und seinem Wesen.

Künstliche Neuronale Netze stellen den Versuch dar, die Funktionsweise und Leistungsfähigkeit von natürlichen Gehirnen nachzubilden. MCCULLOCH und PITTS [MP43] legten dafür 1943 mit ihrem Modell eines Neurons als logisches Schwellwertelement den Grundstein. Ein solches Element verfügt über mehrere Eingangssignale, deren Wert aufsummiert wird. Sobald dieser Wert eine festgelegte Grenze überschreitet, ist das Neuron aktiv, andernfalls nicht. Der Ausgabewert ist dabei digital codiert, wobei der Ausgabewert 1 beträgt, sobald das Neuron erregt ist und 0, falls nicht.

In den meisten heute verwendeten Modellen ist die Ausgabe nicht unbedingt digital codiert. Es sind vielmehr unterschiedlichste Funktionen denkbar. Verbreitet sind beispielsweise lineare, stufenförmige oder sigmoide Funktionen. Je nach Anwendungsfall ist eine geeignete Funktion auszuwählen. In der vorliegenden Arbeit wird eine sigmoide Funktion verwendet, die den Definitionsbereich von  $-\infty$  bis  $+\infty$  auf den Wertebereich von 0 bis 1 abbildet. Dieses Verhalten kommt dem eines natürlichen Neurons am nächsten. [Nag10, Pat08]

In Abbildung 2.1 wird ein natürliches Neuron dem Modell von MCCULLOCH und PITTS gegenübergestellt.

ROSENBLATT [Ros58] stellte 1958 das Modell des Perzeptrons vor, welches bis heute die Grundlage der Künstlichen Neuronalen Netze ist. In einem Perzeptron sind die Neuronen in Schichten angeordnet und es werden

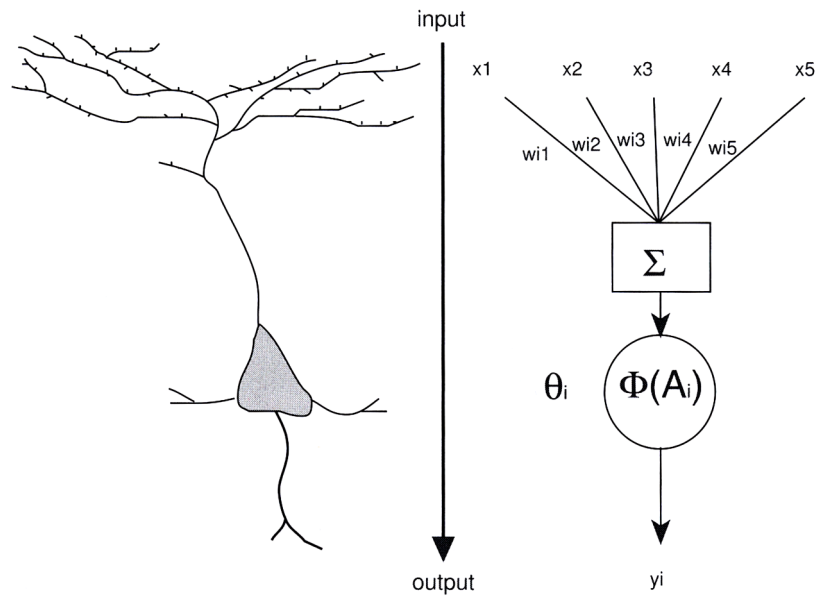


Abbildung 2.1.: Gegenüberstellung der schematischen Darstellung eines Neurons mit dem Modell von MCCULLOCH und PITTS aus [FM08]. Im natürlichen Neuron sind der Dendritenbaum, der Zellkern und das Axon erkennbar. Im Modell erhält das Neuron die gewichteten Eingabewerte, summiert diese auf und gibt den Ausgabewert in Abhängigkeit einer Funktion aus.

veränderbare Synapsengewichte modelliert, die einen Lernprozess ermöglichen. Aus Gründen der Einfachheit wird in dieser Arbeit nicht genauer auf Perzeptronen eingegangen, sondern auf das Modell der Neuronen nach MCCULLOCH und PITTS zurückgegriffen. Näheres zu Perzeptronen findet sich u.a. in [RMS91, Ros58].

Wie der Name schon andeutet, sind in einem Künstlichen Neuronalen Netz mehrere Neuronen miteinander verbunden. Die einzelnen Neuronen werden in der Regel in Schichten zusammengefasst. In der Eingabeschicht befinden sich jene Neuronen, die ihre Eingabewerte aus der Umwelt, zum Beispiel von Sensoren, erhalten. Ausgabeneuronen geben Ausgabewerte an die Umwelt weiter und sind in der Ausgabeschicht zu finden. Die innere Schicht, auch *hidden layer* genannt, umfasst die anderen Neuronen, die weder Eingabe- noch Ausgabeneuronen sind. Abbildung 2.2 zeigt ein einfaches, nicht-rekurrentes Künstliches Neuronales Netz mit einer inneren Schicht. Auf die Bedeutung von Rekurrenz wird im folgenden Abschnitt eingegangen.

Die in dieser Arbeit verwendeten Netze bauen auf sogenannten FAHLMANN-Netzen auf. [FL90, Fah91] Die Topologie eines solchen Netzes lässt sich durch eine Gewichtsmatrix beschreiben. Diese Art der Beschreibung ermöglicht einen guten Überblick über die Verbindungen zwischen den Neuronen.

Das Element  $a_{ij}$  der Matrix gibt dabei das Gewicht der Verbindung eines Neurons  $j$  zu einem Neuron  $i$  an. Die so aufgebaute Matrix enthält alle Verbindungen eines Netzes. Ein bestimmtes Neuron ist durch genau eine Spalte und genau eine Zeile dargestellt. In allen Künstlichen Neuronalen Netzen dieser Arbeit ist ein spezielles Neuron, Bias-Neuron genannt, enthalten. Dieses besitzt keinen Eingang und gibt stets den Wert 1 aus. Es dient damit der konstanten Erregung des Netzes. In der Matrix stellen die erste Spalte und Zeile das Bias-Neuron dar. Die nächsten Spalten und Zeilen repräsentieren die innere Schicht. Zuletzt kommen die Ausgabeneuronen. Auf diese Weise lässt sich das gesamte Netz mit allen Verbindungen und jeweiligen Gewichten als Adjazenzmatrix darstellen.

Abbildung 2.3 zeigt ein solches Künstliches Neuronales Netz. Das ers-

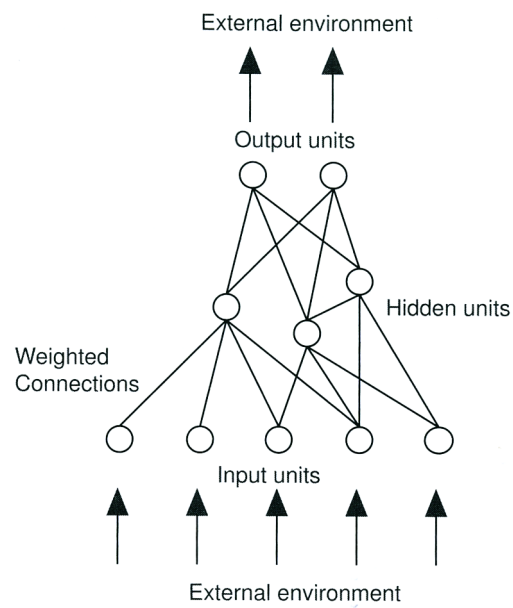


Abbildung 2.2.: Ein nicht-rekurrentes Künstliches Neuronales Netz mit Eingabe-, Ausgabe- und inneren Neuronen aus [FM08]. Es sind beliebig viele Ebenen von inneren Neuronen möglich.

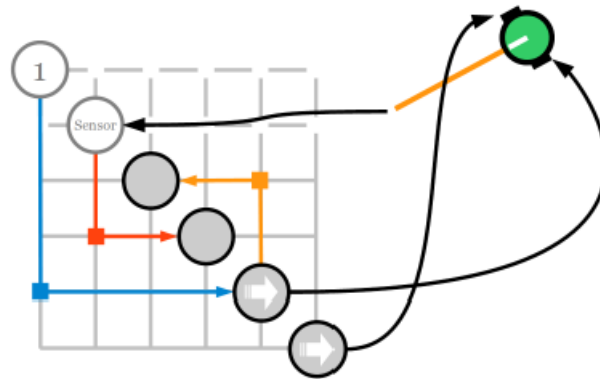


Abbildung 2.3.: Matrixdarstellung eines Künstlichen Neuronalen Netzes aus [Mü10].

te Neuron ist das Bias-Neuron, welches stets 1 als Ausgabe liefert. Das nächste Neuron ist das einzige Eingabeneuron, welches Werte vom Sensor erhält. Die nächsten beiden Neuronen sind innere Neuronen. Bei den letzten beiden handelt es sich um Ausgabeneuronen. Die Quadrate auf den Gitterlinien definieren die Verbindungen zwischen den Neuronen. Es ist zu erkennen, dass das Eingabeneuron mit dem zweiten inneren Neuron verbunden ist (rot). Das Bias-Neuron ist mit dem ersten Ausgabeneuron verbunden (blau). Diese Synapsen sind vorwärtsgerichtet. Oberhalb der Diagonale der Matrix befindet sich ein weiteres Quadrat (orange). Diese Verbindung ist rückwärtsgerichtet und verbindet das erste Ausgabeneuron mit dem ersten inneren Neuron. [Mü10]

Die in dieser Arbeit verwendeten Künstlichen Neuronalen Netze stammen aus den Arbeiten von MÜLLER, NAGEL und COLLING. Weitere Ausführungen zu den Netzen würden den Rahmen dieser Arbeit sprengen und sind deshalb dort zu finden. [Col11, Mü10, Nag10]

Das Zusammenspiel der Neuronen ermöglicht es, komplexe Schaltvorgänge durchzuführen und aufwendige Steuerungsaufgaben zu übernehmen. Am Beispiel des menschlichen Gehirn lässt sich die Mächtigkeit Neuronaler



Netze erraten. Gegenüber anderen Steuerungsverfahren haben Künstliche Neuronale Netze den Nachteil, dass ihre Funktionsweise nur schwer nachvollzogen oder interpretiert werden kann. So ist selbst bei kleinen Netzen in der Regel keine Aussage über das Verhalten möglich. [Nag10]

### 2.3.3.2. Feed-Forward-Netze und rekurrente Netze

Das oben vorgestellte Modell eines Künstlichen Neuronalen Netzes baut auf durch Synapsen verbundenen Neuronen auf. Es ist zu unterscheiden, welche Neuronen miteinander verbunden werden dürfen. In sogenannten Feed-Forward-Netzen dürfen Neuronen aus tieferen Schichten nicht mit Neuronen aus höheren Schichten verbunden werden. Beispielsweise kann ein Ausgabeneuron nicht mit einem Neuron aus der inneren Schicht verbunden werden. Dies bedeutet, dass Rückkopplungen nicht möglich sind.

In rekurrenten Netzen ist es hingegen gestattet, die Neuronen beliebig miteinander zu verbinden. Somit ist es auch möglich, Neuronen mit sich selbst oder Ausgabeneuronen mit inneren Neuronen zu verbinden. Diese zusätzlichen Synapsen erweitern das Verhalten eines Netzes. Durch sie können bei der Verarbeitung Zeit, Eingabesequenzen, Oszillationen und Mehrdeutigkeiten der Eingaben berücksichtigt werden. [CS97] Weiterhin kann gezeigt werden, dass rekurrente Netze turing-vollständig sind. [Hyö96]

Gehirne von Lebewesen sind im Vergleich zu Künstlichen Neuronalen Netzen mit einer um viele Größenordnungen höheren Zahl an Neuronen ausgestattet. Die Neuronen sind ebenfalls so verbunden, dass Rückkopplungen möglich sind. Dies ermöglicht dem Menschen das Erkennen von Mustern, gibt ihm das Gedächtnis und lässt ihn lernen. Der Vorgang des Lernens und das Gedächtnis sind jedoch deutlich komplexer als bei Künstlichen Neuronalen Netzen. Näheres hierzu findet sich in [Dow04] und [MK98].

In der vorliegenden Arbeit werden sowohl reine Feed-Forward-Netze als auch rekurrente Netze verwendet. In den später durchgeführten Experimenten wird untersucht, ob die höhere Mächtigkeit rekurrenter Netze einen Vorteil bei der Steuerung der Agenten mit sich bringt oder nicht.

### 2.3.3.3. Evolution statt Lernverfahren

Die Möglichkeiten, Künstliche Neuronale Netze gezielt zu erstellen, sind stark beschränkt. Das Verhalten eines bestimmten Netzes ist nicht oder nur schwer vorherzusagen. Dadurch ist die Konstruktion eines Netzes, welches zu gewissen Eingaben bestimmte Ausgaben liefern soll, in der Regel nicht möglich. Dies gilt insbesondere auch in der vorliegenden Arbeit.

Wenn Künstliche Neuronale Netze nicht gezielt erstellt werden können, dann müssen sie angepasst werden. Eine Möglichkeit dieser Anpassung ist das Lernen. Es existieren verschiedene Lernverfahren, welche angewendet werden können, um Netze zu trainieren. Beispielhaft seien hier *Backpropagation* und *Cascade-Correlation* genannt. [FL90, Fah91] Den Lernverfahren ist gemeinsam, dass ein bestimmter Trainingssatz an Eingabe- und zugehörigen Ausgabedaten verfügbar sein muss. Die Eingabedaten werden dem Netz übergeben, woraufhin die Ausgabedaten erzeugt werden. In Abhängigkeit der errechneten Ausgabedaten werden die Gewichte der Synapsen angepasst, bis das Netz die gewünschten Ausgaben erzeugt.

In der vorliegenden Arbeit waren Lernverfahren nicht anwendbar. Bei Problemen wie dem Erkennen von Morse-Code, Bildern, Buchstaben usw. sind zu den verschiedenen Eingaben stets die erwünschten Ausgaben verfügbar. Das Fehlen der erwarteten Ausgaben stellt das Problem dar, wegen welchem im Rahmen dieser Arbeit keine Lernverfahren möglich waren. Es ist nicht möglich, zu sagen, welche Ausgaben „richtig“ sind, den Agenten also erfolgreich steuern. Die zu einer bestimmten Eingabe passenden Ausgabewerte sind schlichtweg nicht bekannt. Aus diesem Grund werden neben den Körpern auch die Künstlichen Neuronalen Netze der Agenten durch Evolutionäre Algorithmen optimiert.

## 2.4. Verwandte Arbeiten

In diesem Abschnitt wird eine Auswahl von Arbeiten vorgestellt, die mit den hier durchgeführten Untersuchungen verwandt sind.

### 2.4.1. Pionierleistungen von Sims

SIMS [Sim94a, Sim94b] leistete auf dem Gebiet der Evolutionären Robotik Pionierarbeit. In seinen Veröffentlichungen von 1994 stellt er ein System zur Evolution dreidimensionaler Wesen vor, wobei er gerichtete Graphen als Genome für die Wesen verwendet. In diesen Graphen sind sowohl die Eigenschaften des Körpers als auch des Gehirns eines Wesens festgelegt. Die Verwendung von Graphenstrukturen in Genomen erlaubt fraktale Agenten, da Rekursion und Symmetrie unterstützt werden.

Die Knoten des Graphen beschreiben jeweils einen Körperteil mit einem festgelegten Gelenktyp. Des Weiteren verfügt jeder Knoten über lokale Neuronen, die zusammen ein Künstliches Neuronales Netz ergeben. Agenten können mit Sensoren und Aktoren ausgestattet sein, die es ihnen gestatten, Informationen aus der Umwelt zu erfassen bzw. mit der Umwelt zu interagieren. Sie werden über ein zentrales Künstliches Neuronales Netz gesteuert.

Die Evaluation der Agenten erfolgt in einer physikalisch korrekt simulierten Welt, wobei neben der Fortbewegung an Land auch die Simulation einer Unterwasserwelt möglich ist. SIMS führte eine Reihe von Experimenten durch. In manchen ist die kontrollierte Fortbewegung an Land oder im Wasser das Ziel der Evolution. In anderen müssen konkurrierende Agenten die Kontrolle über einen Würfel erhalten, um einen hohen Fitnesswert zu erreichen. In wiederum anderen ist das Verfolgen einer Lichtquelle Ziel der Untersuchungen.

Alle Experimente brachten erfolgreiche Individuen hervor, die die gestellten Aufgaben erfüllen können. So konnten beispielsweise verschiedene Arten von fortbewegungsfähigen Agenten evolviert werden. Ebenso konnten Agenten, welche Lichtquellen folgen, hervorgebracht werden. Auch das Ringen um die Kontrolle über einen Würfel wurde erfolgreich gelöst.

In den Veröffentlichungen wird darauf hingewiesen, dass durch die Evolution der Agenten Fehler in der Physik-Engine ausgenutzt werden. Diese Beobachtung konnte auch in den für die vorliegende Arbeit durchgeführten Experimenten gemacht werden.

In beiden Arbeiten von SIMS wird die Simulationsumgebung nur spärlich beschrieben. Das Simulationswerkzeug ist ebenso wenig öffentlich verfügbar wie die Simulationsdaten, weshalb die Experimente nicht direkt nachvollziehbar sind. Trotzdem stellen diese Leistungen den Ausgangspunkt für weitere Arbeiten und Forschungen dar.

Die vorliegende Arbeit wurde ebenfalls von den Arbeiten von SIMS inspiriert.

### 2.4.2. Evolutionary Robotics von Nolfi und Floreano

NOLFI und FLOREANO schlagen mit ihrem Werk *Evolutionary Robotics* [NF01] den Bogen von der simulierten Evolution autonomer Roboter zur technischen Realisierung. Sie verwenden einfache, modular aufgebaute Roboter vom Typ KHEPERA, welche über Infrarot- und teilweise auch Tastsensoren verfügen. Des Weiteren können sie zwei Räder ansteuern, welche eine Fortbewegung ermöglichen. In manchen Experimenten verfügen die Roboter zusätzlich über Greifarme oder erweiterte Sensoren.

Das Verhalten der Roboter wird in verschiedenen Experimenten evolviert. So sind beispielsweise Navigationsaufgaben oder das Sammeln von Gegenständen zu bewältigen. Die Autoren zeigen, dass es unter Verwendung von Künstlichen Neuronalen Netzen und Evolutionären Algorithmen möglich ist, die Problemstellungen zu lösen. Auch das Verhalten komplexer Roboter, welche sich durch die gezielte Bewegung von Gliedmaßen fortbewegen sollen, wird untersucht.

Zuletzt diskutieren NOLFI und FLOREANO evolvierbare Hardware. Dabei handelt es sich um *Field Programmable Gate Arrays*, die ihre Verschaltung durch Evolution verändern. Es wird festgestellt, dass das Gebiet vielversprechend ist, aber die nötige Technologie zum damaligen Zeitpunkt nicht (öffentlich) verfügbar war.

Im Vergleich zur vorliegenden Arbeit erfolgen die meisten Versuche bei NOLFI und FLOREANO auf sich nicht verändernden, realen Robotern. Diese werden zunächst aufgebaut und anschließend wird das gewünschte Verhalten entwickelt. In der vorliegenden Arbeit liegt der Fokus hingegen auf

simulierten Robotern, bei welchen sowohl Körperstrukturen als auch das Gehirn des Agenten gleichzeitig verändert werden.

### 2.4.3. Erweiterung der Arbeiten von Sims durch Krčah

In [Krč07] greift KRČAH die Experimente von SIMS auf. Die Veröffentlichung beschreibt das Erstellen und Testen einer Simulationsumgebung, in welcher die Versuche von SIMS nachvollzogen werden können. Da die von SIMS verwendete Implementierung nicht öffentlich gemacht wurde, erstellte KRČAH ein neues Simulationswerkzeug. Es wird gezeigt, dass es damit möglich ist, die Arbeiten von SIMS zu wiederholen und darauf aufbauend Experimente durchzuführen. Beispiele sowie Videos aus diesen sind unter [Krč11] öffentlich verfügbar.

In den durchgeführten Versuchsreihen fiel, wie auch in dieser Arbeit, auf, dass gewisse Individuen die Eigenheiten der Physik-Engine ausnutzen, um eine besonders hohe Fitness zu erreichen. So entwickelten sich Wesen, die sich der Besonderheiten eines Scharniergelenks bedienen, um eine Art Propeller zu bilden. Dieser Propeller verhilft den Agenten dann durch seine Bewegung zu einer unerwartet hohen Fitness.

KRČAH stellt in [Krč08] einen auf NEAT<sup>2</sup> aufbauenden neuen Algorithmus vor. Dieser wird HIERARCHICAL NEAT genannt und führt – wie der Name schon suggeriert – eine Hierarchie in den Algorithmus ein. Die Hierarchie ermöglicht eine bessere Performance im Vergleich zu anderen Genetischen Algorithmen und garantiert einen gewissen Grad an Diversität in einer Population. Es wurden Experimente mit mehr als acht Monaten Rechenzeit durchgeführt, in welchen dies gezeigt wurde.

Die Veröffentlichungen [KT10] und [Krč10] präsentieren einen weiteren Algorithmus zur gleichzeitigen Evolution von Körper und Gehirn. In diesem wird nicht die Fitness eines Agenten zur Bewertung herangezogen, sondern sein Verhalten auf Einzigartigkeit untersucht. Nur die Agenten, die neues, bisher noch nicht beobachtetes Verhalten zeigen, dürfen sich fortpflanzen.

---

<sup>2</sup>NEAT steht für *NeuroEvolution of Augmenting Topologies*. Näheres zu diesem Algorithmus findet sich unter [Sta11]. Für eine Anwendung im Bereich Evolutionäre Robotik siehe beispielsweise [Pat08].

Es wird gezeigt, dass der Algorithmus in der Lage ist, Probleme wie das Umgehen einer Barriere zu lösen, während rein fitnessbasierte Algorithmen dies nicht können. Aber auch in normalen Experimenten ohne irreführende Umgebung zeigt der vorgeschlagene Lösungsweg eine vergleichsweise gute Leistung.

### 2.4.4. Arbeiten von Lehmann und Stanley

Das von KRČAĤ entwickelte Simulationswerkzeug wird in [LS11] als Grundlage für weitere Experimente verwendet. Die Autoren implementieren einen Evolutionären Algorithmus, welcher eine lokale Suche mit einer Suche nach bisher unbekanntem Verhalten kombiniert. Dabei wird die Bildung von einzelnen Arten von Individuen, die jeweils Nischen ausnutzen, forciert. Dies hat wiederum eine hohe Diversität der Population zur Folge. Der Algorithmus verhindert so verfrühte Konvergenz und lokale Optima.

### 2.4.5. Erweiterung der Arbeiten von Sims durch Lassabe et al.

Auch LASSABE ET AL. [LLD07] greifen die Ideen von SIMS auf. Sie verwenden jedoch anstatt eines Künstlichen Neuronalen Netzes ein *Classifier System* als Steuerungseinheit der Agenten. In diesem Bereich weicht die Arbeit stark von den hier gewählten Ansätzen ab. Sie sei trotzdem erwähnt, da die durchgeführten Simulationen nicht in einfachen Räumen sondern vielmehr in komplexeren Welten ablaufen. Die Arbeit diskutiert u.a. auch die Evolution eines gesamten Ökosystems mit verschiedenen Agenten und Pflanzen.

### 2.4.6. Künstliche Neuronale Netze nach Fahlmann

Die in dieser Arbeit verwendeten Künstlichen Neuronalen Netze basieren auf den Arbeiten von FAHLMANN. In [FL90, Fah91] werden mit *Cascade-Correlation* eine Architektur und ein Lernverfahren vorgestellt, welche es ermöglichen, Künstliche Neuronale Netze effizient zu trainieren. Abweichend zur vorliegenden Arbeit sind die gewünschten Ausgaben zu vorgegeben Eingaben jedoch bekannt. So wird beispielsweise das Lernen von

Morse-Code durch ein Netz beschrieben. Da bei der Evolution dreidimensionaler Agenten die gewünschten Ausgaben aber nicht bekannt sind, kann auch kein Lernverfahren angewendet werden. Die Struktur der hier verwendeten Netze basiert jedoch auf den Arbeiten, weshalb sie hier erwähnt sein sollen.

### 2.4.7. Arbeiten von Helaoui

In seiner Arbeit verwendet HELAOUI [Hel08] *Gene Regulatory Networks*<sup>3</sup>, um eine spezielle Form von Agenten zu evolvieren. Die Agenten bestehen aus Kugeln, welche sich mit einer bestimmten Geschwindigkeit drehen oder fix sein können. Es wird gezeigt, dass die Entwicklung sich fortbewegender Agenten durch Evolution möglich ist. Dabei werden auch komplexere Aufgaben wie das Überwinden von Hindernissen gelöst. Die Entwicklung der Agenten wird immer im Hinblick auf Parallelen zur Natur durchgeführt, so dass die Experimente auch aus biologischer Sicht verteidigt werden können.

Im Vergleich zu der vorliegenden Arbeit unterscheiden sich die Agenten. Während bei HELAOUI kugelförmige, sich kontinuierlich drehende Bausteine verwendet werden, kommen hier quaderförmige Gliedmaßen zum Einsatz, die über Gelenkbewegungen von einem Künstlichen Neuronalen Netz gesteuert werden.

### 2.4.8. Arbeiten über Künstliche Neuronale Netze von Müller, Nagel und Colling

In [Mü10] beschreibt MÜLLER die Implementierung der FAHLMANN-Netz-Architektur in das EAS-Framework. Die in der Studienarbeit verwendeten Netze stellen die Basis der Gehirne der Agenten dieser Arbeit dar. Die Künstlichen Neuronalen Netze werden in [Mü10] verwendet, um simulierte Roboter zu steuern. So werden Putzroboter evolviert, die einen

---

<sup>3</sup>Zu deutsch etwa Genregulierende Netzwerke. Genregulation bezeichnet die Steuerung der Aktivität von Genen. Vereinfacht gesagt kann so festgelegt werden, ob bestimmte Gene oder Bereiche eines Genoms aktiv sind oder nicht. Vgl. hierzu auch [Egg97] und [KB03].

mit Hindernissen versehenen, simulierten Raum erfolgreich reinigen können. Außerdem konnten Roboter entwickelt werden, die in Mannschaften gegeneinander antreten, um einen Ball ins gegnerische Tor zu befördern.

NAGEL [Nag10] verwendet die implementierten Netze, um die Kooperation von Roboterschwärmen zu untersuchen. Dazu wird eine Reihe von Agenten mit Künstlichen Neuronalen Netzen versehen, um anschließend die Evolution dieser Agenten in einem Räuber/Beute-Szenario zu analysieren.

COLLING [Col11] erweitert die Künstlichen Neuronalen Netze des EAS-Frameworks und führt mit diesen Physik-Experimente durch. Er implementiert Agenten, die in einer physikalisch korrekt simulierten 2D-Umgebung eine stabile Umlaufbahn um einen Planeten entwickeln sollen. Es gelingt zu zeigen, dass unter verschiedenen Bedingungen durch Evolution solche Agenten entstehen, die überlebensfähig sind, da sie eine stabile Umlaufbahn erreichen und beibehalten können. Wie auch in [Mü10] und [Nag10] werden die Körper der Agenten nicht verändert. Die Evolution findet lediglich auf Ebene des Künstlichen Neuronalen Netzes statt.

Die in den drei Arbeiten implementierten und entwickelten Klassen dienen als Grundlage für die hier verwendeten Künstlichen Neuronalen Netze. Die ursprünglichen Klassen konnten stellenweise unverändert übernommen werden, mussten aber andererseits in manchen Bereichen angepasst und erweitert werden.



## 3. Entwurf und Implementierung

Einfachheit ist die höchste  
Stufe der Vollendung.

---

*Leonardo Da Vinci*

Dieses Kapitel stellt den Kern der vorliegenden Arbeit dar. Der Entwurf und die Implementierung der Simulationsumgebung und der Agenten war ein Prozess, welcher schrittweise vollzogen wurde. Deshalb wird zu Beginn dieses Kapitels die Erstellung der Simulationsumgebung mit der enthaltenen Physik-Engine beschrieben. Anschließend werden der entworfene Agent, die Simulationswelt und weitere für die Experimente nötige Klassen vorgestellt und erklärt.

### 3.1. Erstellung einer Simulationsumgebung

Es stehen diverse Simulationswerkzeuge zur Verfügung, mit welchen die Evolution von Agenten modelliert und simuliert werden kann. Für diese Arbeit wurde das am Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB) des Karlsruher Instituts für Technologie (KIT) entwickelte EAS-Framework verwendet.

#### 3.1.1. Das EAS-Framework

Das EAS-Framework [KP11] wurde von LUKAS KÖNIG und DANIEL PATH-MAPERUMA entwickelt. Das Akronym EAS hat zwei Bedeutungen. Zuerst

steht es für *Easy Agent Simulation*. Daran lässt sich der Zweck des Frameworks bereits erkennen: die einfache Simulation von Agenten. Weiter lassen sich aus der Abkürzung EAS auch die drei wichtigsten Komponenten einer Simulation in diesem Framework ableiten: das *Environment*, die Agenten und der *Scheduler*.

Im Folgenden soll kurz auf die prinzipielle Funktionsweise des Frameworks und der damit möglichen Simulationen eingegangen werden. Wie bereits angedeutet, sind für eine Simulation zumindest ein *Environment*, ein Agent und ein Scheduler erforderlich. Bei dem *Environment* handelt es sich um eine Umgebung, in welcher die Agenten simuliert werden. Diese Umgebung kann beispielsweise ein Koordinatengitter sein, welches die Position der Agenten beschreibt. Eine zweidimensionale, physikalisch korrekte Simulation ist aber ebenso möglich. Je nach Simulationsanforderung ist ein geeignetes *Environment* auszuwählen bzw. zu erstellen. Im Rahmen dieser Diplomarbeit war es nötig, dreidimensionale Physik-Simulationen durchzuführen. Das EAS-Framework stellte zu Beginn kein *Environment* zur Verfügung, welches physikalisch korrekte Simulationen in einer dreidimensionalen Welt ermöglichte. Deshalb musste ein komplett neues *Environment* programmiert werden.

Bei den Agenten handelt es sich um die eigentlich zu untersuchenden Objekte. Ein Agent kann beispielsweise ein Roboter sein, dessen Verhalten untersucht werden soll. Auch hier gilt, wie bei den *Environments*, dass ein passender Agent ausgewählt oder erstellt werden muss. Da anfangs noch kein passendes *Environment* verfügbar war, waren auch keine Agenten vorhanden. Somit mussten nach dem Erstellen der Umgebung auch passende Agenten programmiert werden.

Die Agenten agieren im *Environment* und werden vom Scheduler getaktet. Der Scheduler dient der Steuerung des Simulationsablaufs. Durch ihn wird die zeitliche Abfolge der Simulationsschritte und deren Inhalt definiert. Mit dem Scheduler werden Agenten erzeugt, dem *Environment* hinzugefügt und darin simuliert. Er ermöglicht außerdem die für diese Arbeit nötige Evolution von Agenten. Was für das *Environment* und die Agenten wichtig ist, gilt auch in besonderem Maße für die Scheduler: sie müssen an

die jeweilige Aufgabe angepasst bzw. speziell dafür erstellt werden. Weitere optionale Bestandteile des Frameworks sind Plugins, Aktoren und Sensoren, die die Funktionalitäten des Frameworks, der Environments oder der Agenten erweitern können.

Auf diesem Grundgerüst aufbauend galt es, neue Komponenten zu erstellen, so dass die für diese Arbeit nötigen Simulationen durchgeführt werden können. Dazu musste zuerst eine passende 3D-Physik-Engine ausgewählt werden.

#### 3.1.2. Auswahl einer 3D-Physik-Engine

Das EAS-Framework war zu Beginn dieser Arbeit nicht 3D-fähig, da kein dreidimensionales Environment vorhanden war. Es ermöglichte insbesondere keine *physikalisch korrekten* Simulationen in einer dreidimensionalen Welt.

Eine 2D-Physik-Engine war zwar bereits implementiert und funktionsfähig und dazu passende Environments waren ebenfalls verfügbar. Die 2D-Physik-Engine ist jedoch nicht in der Lage, dreidimensionale Berechnungen durchzuführen. Somit ließen sich auch die bestehenden Environments weder anpassen noch erweitern. Aus diesem Grund war die Erstellung eines passenden Environments, welches denen durch diese Arbeit entstandenen Anforderungen genügt, vonnöten. Das Environment sollte realistische Physik-Simulationen in einer dreidimensionalen Welt ermöglichen. Dafür ist eine 3D-Physik-Engine nötig, welche vom Environment genutzt werden kann, um die nötigen Berechnungen durchzuführen.

##### 3.1.2.1. Anforderungen an eine 3D-Physik-Engine

Bei der Auswahl der Engine waren folgende Anforderungen zu beachten:

- Das neu erstellte Environment und die zugrundeliegende Engine sollen auch in weiteren Projekten und Arbeiten verwendet werden können. Dazu muss das Erstellen einer (abstrakten) Environment-Klasse möglich sein, aus welcher sich wiederum andere Klassen ableiten lassen, die speziell auf die jeweiligen Anforderungen zugeschnitten sind.

Es darf keine strikte Festlegung auf einen alleinigen Anwendungsfall geben.

- Neben der hohen Flexibilität und Anpassbarkeit ist es auch ein Ziel, die Verwendung möglichst einfach zu gestalten. Die erforderliche Einarbeitung in die 3D-Physik-Engine soll möglichst reduziert werden, d.h. neue Projekte sollen mit einfach zu erstellenden Klassen, die ggf. von bereits vorhandenen Klassen abgeleitet werden können, realisierbar sein. Ein tieferes Verständnis der 3D-Physik-Engine darf nicht vorausgesetzt werden. Stattdessen muss die Verwendung der passenden Environment-Klassen ausreichend sein, um physikalisch korrekte Simulationen durchführen zu können.
- Da Physik-Simulationen in der Regel viel Rechenzeit beanspruchen, ist es ein Ziel, diese auf einem verteilten Rechnernetz durchführen zu können. Dazu steht am KIT das Job Scheduling Karlsruhe (JO SCHKA) [Bon07, Bon08] zur Verfügung. Mit Hilfe dieses Werkzeugs sind Simulationen auf mehreren verteilten Rechnern möglich, wobei die Steuerung der Simulation zentral erfolgen kann. Um diese Möglichkeit einfach und problemlos nutzen zu können, muss die Engine folgende Anforderungen erfüllen:
  - Die gesamte Simulationsumgebung sollte ausschließlich auf Java basieren, da nur so eine hohe Betriebssystemunabhängigkeit gewährleistet ist. Die Verwendung von eigenen Laufzeitbibliotheken, die nicht in Java vorliegen, muss vermieden werden.
  - Sämtliche für die Simulation nötigen Dateien müssen in einem einzigen Verzeichnis vorliegen. Es sind ausschließlich flache Hierarchien gestattet, geschachtelte Ordnerstrukturen sind nicht möglich.
  - Während der Simulation dürfen sich keine Dialoge, Fenster oder Ähnliches öffnen, da dies auf den verwendeten Rechnern zu einer Störung führen würde. Dies erfordert insbesondere eine Trennung von Simulation und Grafik, d.h. die 3D-Physik-Engine

muss Berechnungen ohne gleichzeitige grafische Darstellung ermöglichen.

- Um die Ergebnisse der Simulation besser präsentieren zu können, war eine grafische Darstellung wünschenswert. Sie sollte bei Bedarf zuschaltbar sein, die eigentliche Simulation muss aber ohne sie ablaufen können.
- Da eine bestehende 3D-Physik-Engine nicht unverändert übernommen werden kann, sondern angepasst werden muss und außerdem eine Verwendung in anderen Projekten wünschenswert ist, muss sie einer geeigneten Lizenz unterliegen. Diese Lizenz muss eine Veränderung und Einbindung in andere Projekte erlauben.

#### 3.1.2.2. JBullet als 3D-Physik-Engine

Zum Zeitpunkt der Implementierung der 3D-Physik-Engine in das EAS-Framework gab es nur eine Engine, welche die oben genannten Anforderung erfüllt: JBULLET [Dvo10a].

JBULLET ist eine Java-Portierung der weit verbreiteten Physik-Bibliothek BULLET. Während BULLET in C geschrieben ist, wurde JBULLET von MARTIN DVORAK in Java übertragen. Der gesamte Quellcode steht als Open-Source in reinem Java zur Verfügung. Dabei wurden viele – wenn auch nicht alle – Features von BULLET in Java übertragen. JBULLET ermöglicht neben einer Kollisionserkennung auch eine physikalisch korrekte Simulation einer dreidimensionalen Umgebung mit verschiedenen Constraints. Dabei werden verschiedenste Kollisionsformen (Kugel, Quader, Kegel, ...) sowie Gelenke (Scharnier, Kugelgelenk, ...) unterstützt. Außerdem liegt JBULLET unter einer geeigneten Lizenz, der ZLIB Lizenz, vor. Diese Lizenz besagt Folgendes. [Ope11]

This software is provided ,as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

**Vorteile von JBullet** Offensichtlich schränkt die Lizenz die Verwendung von JBULLET kaum ein. Sie erlaubt sämtliche Aspekte, die für diese Arbeit und kommende Projekte nötig sind. Die Software darf für alle Anwendungen verwendet werden, eine Änderung ist ebenso möglich wie eine Verbreitung. Damit ist JBULLET geradezu prädestiniert für die Verwendung im EAS-Framework.

- Ein beliebiges Anpassen sowie ein Ableiten von eigenen Klassen ist dadurch möglich, dass JBULLET komplett in Java vorliegt und die Lizenz dies gestattet. So lassen sich (abstrakte) Klassen erstellen, welche die Anzahl an nötigen Low-Level-Engine-Befehlen reduzieren und eine Verwendung deutlich vereinfachen. Trotzdem ist eine enorme Anpassbarkeit gegeben, da eigene Klassen beliebig von den bestehenden abgeleitet werden dürfen.
- Der Quell-Code von JBULLET liegt komplett in Java vor und die gesamte Bibliothek kann in einem einzelnen Verzeichnis liegen. Dazu musste zwar der Build-Path in ECLIPSE angepasst werden, aber die einzelnen Klassen an sich können unverändert übernommen werden.<sup>1</sup>

---

<sup>1</sup>Das Ändern des Build-Paths könnte potentiell zu Beeinträchtigungen von anderen Projekten im EAS-Framework führen, da die Reihenfolge beim Kompilieren angepasst werden musste. Probleme oder negative Auswirkungen wurden aber nicht festgestellt.

Dies gewährleistet neben der problemlosen Ausführung der Experimente mit JOSCHKA auch eine einfache Wart- und Aktualisierbarkeit. Betriebssystemabhängige Klassen sind lediglich für die grafische Darstellung nötig. Da diese aber unabhängig von der Simulation erfolgt, stellt dies kein Problem dar.

- Die physikalische Simulation erfolgt bei JBULLET komplett getrennt von einer grafischen Darstellung. So lassen sich die eigentliche Simulation und die Präsentation unabhängig voneinander durchführen und eine Zuschaltung der Grafik erfolgt nur dann, wenn dies auch gewünscht ist. Der JBULLET-Bibliothek sind diverse Beispiellassen beigefügt, welche eine auf OpenGL basierende Grafik-Engine beinhalten. Nicht zuletzt wegen der geeigneten Lizenz war es möglich, daraus ein Plugin für das EAS-Framework abzuleiten bzw. weiterzuentwickeln, welches bei Bedarf zugeschaltet werden kann, um eine dreidimensionale Visualisierung der Szene zu ermöglichen.

Alle oben genannten Punkte sprechen klar für die Verwendung von JBULLET als 3D-Physik-Engine im EAS-Framework.

Neben JBULLET gibt es derzeit für Java keine veröffentlichten Alternativen, die einen ähnlichen Funktionsumfang, eine vergleichbare Flexibilität und Anpassbarkeit bieten. Es existieren jedoch Frameworks, die JBULLET verwenden und noch weitere Funktionalitäten zur Verfügung stellen. Ein solches Framework ist die JMONKEYENGINE (JME)<sup>2</sup>. Die Verwendung dieser Engine hätte einerseits den großen Vorteil, dass viele Klassen vorgegeben sind, welche die Durchführung der Physik-Simulation deutlich vereinfachen. Andererseits ist die JMONKEYENGINE damit auch komplexer und weniger flexibel als die reine JBULLET-Variante. Erschwerend kommt hinzu, dass umfangreiche Vererbungshierarchien bestehen, weshalb eine Integration in das EAS-Framework nur mit großen Mühen und vielen Restrukturierungen überhaupt möglich wäre.<sup>3</sup>

---

<sup>2</sup><http://jmonkeyengine.com/>

<sup>3</sup>Java gestattet keine echte Mehrfachvererbung bei Klassen. Eine Klasse kann nicht gleichzeitig von mehr als einer Superklasse erben.

**Nachteile von JBullet** Bei allen Vorteilen gibt es allerdings auch diverse Nachteile und Schwierigkeiten bei der Verwendung von JBULLET:

- Die Portierung von BULLET, welches in C geschrieben ist, zu JBULLET war zum Zeitpunkt der Arbeit noch nicht komplett abgeschlossen. Es gibt viele Methoden und Klassen, die in BULLET vorhanden sind, aber noch nicht in JBULLET implementiert wurden. Dies verkompliziert die Simulation häufig, da vereinfachende Methoden noch nicht zur Verfügung stehen oder bestimmte Klassen komplett fehlen. Diverse Beispiele oder auch Teile der Dokumentation von BULLET verwenden jedoch genau die fehlenden Methoden und Klassen und sind so nicht anwendbar.
- BULLET selbst ist nicht frei von Bugs. Diese Bugs wurden bei der 1:1-Portierung in JBULLET übernommen. JBULLET basiert zum Zeitpunkt der Erstellung der Diplomarbeit auf Version 2.72 von BULLET<sup>4</sup>, wodurch viele Bugs noch nicht behoben wurden. Dies äußerte sich in der vorliegenden Arbeit beispielsweise dadurch, dass Agenten in gewissen Situationen unkontrolliert „zappeln“ oder eigentlich feste Körper durchdringen.
- Des Weiteren ist die Dokumentation von JBULLET in vielen Teilbereichen nicht vorhanden oder äußerst lückenhaft. Ganze Klassen sind undokumentiert und häufig lässt sich die Funktion diverser Methoden nur anhand deren Namen bzw. deren Parameternamen erraten. Allerdings liegt der gesamte Quellcode für JBULLET offen vor und aus Beispielen (für JBULLET bzw. BULLET) lässt sich häufig auf die korrekte Anwendung schließen.
- Selbst einfachste Simulationen in JBULLET erfordern standardmäßig eine Vielzahl von Befehlen. Diese Anzahl steigt mit der Komplexität der Simulation nochmals deutlich an. Da eine leichte Verwendung eines der Hauptziele ist, war es nötig, eigene Klassen zu schreiben,

---

<sup>4</sup>BULLET ist in Version 2.79 verfügbar, Stand 30.09.2011.



die die Verwendung deutlich vereinfachen und die nötige Zahl an Befehlen stark reduzieren.

- Zuletzt verwendet JBULLET zur Geschwindigkeitssteigerung eine Byte-Code-Optimierung und einen eigenen Stack.<sup>5</sup> Dies führt normalerweise zu Problemen mit anderen Klassen im EAS-Framework, da der vom Compiler erzeugte Byte-Code verändert wird. Aus diesem Grund musste der Build-Path in ECLIPSE angepasst werden, so dass die Physik-Simulationen, welche JBULLET verwenden, optimiert werden, ohne die anderen Bestandteile des EAS-Frameworks bzw. die anderen Klassen zu beeinflussen.

Die aufgezählten Nachteile und Probleme verhindern nicht eine Verwendung als 3D-Physik-Engine im EAS-Framework. Nicht zuletzt, weil es zu JBULLET keine echten Alternativen gibt, ist diese Engine die beste Möglichkeit, dreidimensionale Physik-Simulationen im EAS-Framework zu ermöglichen.

#### 3.1.2.3. Funktionsweise von JBullet

In diesem Abschnitt soll kurz auf die grundlegende Funktionsweise der 3D-Physik-Engine JBULLET eingegangen werden. Eine 3D-Physik-Engine wie JBULLET wird dazu verwendet, physikalisch korrekte Berechnungen im dreidimensionalen Raum durchzuführen, um eine realistische dreidimensionale Welt simulieren zu können. JBULLET unterstützt bisher lediglich Festkörpersimulationen, während BULLET auch Algorithmen für weiche Körper zur Verfügung stellt. [Bul11, Cou10, Dvo10a, Dvo10b]

In JBULLET werden verschiedenste Kollisionsformen (*CollisionShapes*) angeboten: Ebenen, Quader, Kugeln, Kapseln, Zylinder, Kegel oder konvexe Hüllen. Weiterhin sind aus den vorherigen konvexen Körpern zusammengesetzte Gebilde sowie Mesh-Oberflächen nutzbar, welche dann wiederum nicht konvex sein müssen. Neben den Körpern bilden die Constraints

---

<sup>5</sup>Siehe <http://www.java-gaming.org/index.php/topic,18843.0.html>, abgerufen am 30.09.2011.

genannten Einschränkungen eine weitere wichtige Komponente der Simulation. Zum Zeitpunkt der Erstellung der Diplomarbeit waren nicht alle in BULLET verfügbaren Gelenke bzw. Constraints portiert. Einige der Verfügbaren sind generische Gelenke mit sechs Freiheitsgraden, Kegel-Gelenke, Scharniere und Punkt-Gelenke. Mit diesen Gelenken lassen sich Kollisionskörper verbinden bzw. einschränken, so dass Simulationen zusammenhängender Körper möglich werden.

Im Folgenden soll das Zusammenspiel der verschiedenen Klassen von JBULLET erklärt werden, um die Physik-Simulationen besser verstehen zu können. In den Fußnoten finden sich jeweils die genauen Klassenbezeichnungen, so dass bei Interesse der Quellcode der jeweiligen Klasse betrachtet werden kann. In den meisten Klassen sind zum Zeitpunkt der Erstellung der Diplomarbeit JavaDoc-Kommentare vorhanden.<sup>6</sup>

**Zusammenspiel der JBullet-Klassen** Bevor Simulationen mit JBULLET durchgeführt werden können, muss zunächst eine Simulationsumgebung eingerichtet werden. Diese hat mehrere wichtige Komponenten, die mit ihrem Zusammenspiel die Durchführung der Simulation erst ermöglichen.

Zu Beginn muss eine *CollisionConfiguration*<sup>7</sup> festgelegt werden. Diese ermöglicht es, die Kollisionsalgorithmen zu definieren und zu konfigurieren. In fast allen Fällen ist es ausreichend, eine Standardkonfiguration, die *DefaultCollisionConfiguration*<sup>8</sup> zu verwenden. Diese Standardimplementierung bietet für die meisten Anwendungsbereiche passende Algorithmen an und ist fertig konfiguriert. Sollten weitere, spezielle Algorithmen benötigt werden, dann sind diese explizit zu registrieren. In der vorliegenden Arbeit war dies allerdings nicht der Fall.

Die Klasse *DbvtBroadphase*<sup>9</sup> stellt Methoden zur Verfügung, um potentiell überlappende und damit kollidierende Paare von Kollisionsformen zu finden. Dies ist notwendig, um nicht für alle Paare genaue Berechnung

---

<sup>6</sup>Die gesammelte Dokumentation findet sich auch unter [Dvo10b].

<sup>7</sup>`com.bulletphysics.collision.dispatch.CollisionConfiguration`

<sup>8</sup>`com.bulletphysics.collision.dispatch.DefaultCollisionConfiguration`

<sup>9</sup>`com.bulletphysics.collision.broadphase.DbvtBroadphase`

durchführen zu müssen, obwohl diese möglicherweise gar nicht kollidieren. Um Rechenzeit einzusparen, werden in einer ersten Phase diejenigen Kollisionskörper bestimmt, die sich (im Rahmen einer gewissen Genauigkeit) so nahe sind, dass sie kollidieren könnten. Anschließend können genauere Berechnungen durchgeführt werden. Dazu ist die nächste wichtige Komponente, der *CollisionDispatcher*<sup>10</sup>, nötig, welcher von der Klasse *Dispatcher*<sup>11</sup> erbt.

Während der Simulation werden, wie oben erklärt, zuerst potentiell kollidierende Objekte gesucht. Dies erfolgt in der sogenannten *Broadphase*. Für alle Paare an möglicherweise kollidierenden Objekten erfolgt dann die genauere Berechnung, ob sie tatsächlich überlappen. Sollte dies der Fall sein, werden Zeitpunkt des Zusammenstoßes, die Punkte des Zusammenstoßes und die Penetrationstiefe berechnet. Dem *CollisionDispatcher* kommt hierbei die Aufgabe der Abfertigung der potentiell überlappenden Paare zu. Der *CollisionDispatcher* entscheidet über die Reihenfolge der Abarbeitung und den verwendeten Algorithmus.

Eine weitere für die Simulation notwendige Komponente ist der sogenannte *SequentialImpulseConstraintSolver*<sup>12</sup>. Dieser hat die Aufgabe, die Impulse der Körper zu bestimmen und anzuwenden. Dies wird getan, um Reibung, Elastizität aber auch Einschränkungen bzw. Constraints zu simulieren. Ohne den *SequentialImpulseConstraintSolver* wäre eine Interaktion der Kollisionsformen miteinander nicht möglich.

Zuletzt werden alle vorherigen Komponenten in einer *DiscreteDynamicsWorld*<sup>13</sup> zusammengefasst. Diese wird aus den Teilen *CollisionConfiguration*, *CollisionDispatcher*, *DbtBroadphase* und *SequentialImpulseConstraintSolver* aufgebaut, welche in ihrer Gesamtheit die Simulation einer physikalisch korrekten dreidimensionalen Welt ermöglichen. Wenn eine *DiscreteDynamicsWorld* definiert und erzeugt wurde, können Simulationen in dieser durchgeführt werden. Der Ablauf einer solchen Simulation wird im

---

<sup>10</sup>com.bulletphysics.collision.dispatch.CollisionDispatcher

<sup>11</sup>com.bulletphysics.collision.broadphase.Dispatcher

<sup>12</sup>com.bulletphysics.dynamics.constraintsolver.SequentialImpulseConstraintSolver

<sup>13</sup>com.bulletphysics.dynamics.DiscreteDynamicsWorld

nächsten Unterabschnitt beschrieben.

Die Physik-Welt ist zu Beginn noch komplett leer und somit frei von Kräften, da weder Körper noch eine Gravitation hinzugefügt wurden. Um interagierende Körper simulieren zu können, müssen diese zuerst erstellt und dann der *DiscreteDynamicsWorld* hinzugefügt werden. Die Körper haben ebenso wie die physikalische Welt mehrere Komponenten und Eigenschaften, die kurz vorgestellt werden sollen. Zuerst besitzt ein jeder Körper eine Kollisionsform, auch *CollisionShape*<sup>14</sup> genannt. Bei dieser Form kann es sich um einen einfachen konvexen Körper oder um aufwendige, nicht-konvexe 3D-Modelle handeln. Für einfache, insbesondere für konvexe Körper, sind viele effiziente Algorithmen verfügbar, die eine schnelle Berechnung ermöglichen. Je komplexer und aufwendiger ein Körper aufgebaut ist, desto mehr Rechenzeit ist folglich für die Kollisionsberechnungen einzuplanen.

Neben einer Form besitzen die Körper auch eine Masse. Diese ist gleichmäßig im gesamten Körper verteilt, d.h., er besitzt an allen Stellen dieselbe Dichte. Definitionsgemäß ist ein Körper in JBULLET dynamisch, genau dann wenn seine Masse positiv ist. Eine Masse von null bedeutet, dass der Körper statisch ist und sich nicht bewegen kann. Eine Interaktion mit anderen dynamischen Körpern ist aber sehr wohl möglich. Der Boden einer Simulationsarena beispielsweise muss eine Masse von null besitzen, um in der Welt fixiert zu sein. Sich bewegende Körper können jedoch mit dem Boden kollidieren und von ihm beeinflusst werden.

Dynamische Körper verfügen neben der Masse noch über ein Beharrungsmoment bzw. über eine gewisse Trägheit. Diese kann entweder manuell angegeben oder von der Physik-Engine berechnet werden. Des Weiteren verfügt jeder Körper in der Physik-Welt über eine Position im Raum sowie eine Rotation in selbigem. Diese Informationen werden in einer sogenannten *Transform*<sup>15</sup> gespeichert.

Zu beachten ist bei JBULLET das besondere Koordinatensystem, wel-

---

<sup>14</sup>`com.bulletphysics.collision.shapes.CollisionShape`

<sup>15</sup>`com.bulletphysics.linearmath.Transform`



Abbildung 3.1.: Rechtshändiges Koordinatensystem von JBULLET aus [Cou10]. x- und z-Achse bilden den „Boden“, die y-Achse die Senkrechte. Die x-Achse steht senkrecht auf der Monitorebene.

ches u.a. von üblichen OpenGL-Koordinatensystemen abweicht.<sup>16</sup> Es handelt sich um ein rechtshändiges Koordinatensystem, wobei die y-Achse die Senkrechte darstellt und die x- und z-Achse die Ebene, den „Boden“ bilden. Die x-Achse steht senkrecht auf der Monitorebene, während die z-Achse waagrecht in der Monitorebene liegt. [Cou10] Abbildung 3.1 zeigt die Anordnung der Achsen.

Um während der Simulation Informationen über die Körper auslesen zu können, muss jedes Objekt über einen *DefaultMotionState*<sup>17</sup> verfügen. Mit diesem lassen sich auch zwischen zwei Simulationsschritten (interpolierte) Daten wie Position, Rotation, Geschwindigkeit etc. abfragen. Ohne die *DefaultMotionStates* ist keinerlei Information über die Kollisionskörper während der Simulation abrufbar. Damit ist auch keine grafische Darstellung möglich.

Alle Eigenschaften und Komponenten werden in einer eigenen Klasse, dem *RigidBody*<sup>18</sup> zusammengefasst. Dieser Festkörper wird der Physik-Welt hinzugefügt, in welcher dann die Simulation erfolgen kann. Durch das Zusammenspiel aller Komponenten der *DiscreteDynamicsWorld* und der *RigidBody*-Klassen wird eine physikalisch korrekte Simulation ermöglicht,

<sup>16</sup>Die Koordinatensysteme von JBULLET und der verwendeten Grafik-Bibliothek LIGHTWEIGHT JAVA GAME LIBRARY unterscheiden sich. Eine Umrechnung ist hierbei nötig, um nicht eine spiegelverkehrte Anzeige zu erhalten.

<sup>17</sup>com.bulletphysics.linearmath.DefaultMotionState

<sup>18</sup>com.bulletphysics.dynamics.RigidBody

die auch die Interaktion vieler verschiedener Körper unterstützt. Allerdings sind gewisse Grenzen zu beachten. JBULLET unterstützt nur eine einfache Genauigkeit, wodurch Rundungsfehler relativ schnell zu Tage treten können. Eine Kollision eines sehr kleinen, leichten Körpers mit einem sehr großen, schweren Körper könnte und wird auch in der Regel nicht korrekt simuliert werden. Zu kleine Körper sind ebenso wie zu große Körper zu vermeiden. Dazu ist es gegebenenfalls nötig, die Körper in Größe und Gewicht zu skalieren, um wieder Werte in einem normalen Bereich zu erhalten.

**Ablauf einer Physik-Simulation in JBullet** Sobald alle Körper der Physik-Welt hinzugefügt wurden, kann die eigentliche Simulation beginnen. Dazu sollten, falls gewünscht, zu Beginn noch Eigenschaften wie Elastizität und Reibung der Körper sowie die Gravitation definiert werden. Anschließend kann über die Methode *stepSimulation()* der Klasse *DiscreteDynamicsWorld* die Simulation schrittweise durchgeführt werden. Die Methode *stepSimulation()* sorgt dafür, dass für die gesamte Physik-Welt Berechnungen durchgeführt werden, so dass sie sich anschließend in einem um die in der Methode angegebene Zeit späteren Zustand befinden, sie altert. Hierbei sind dynamische oder statische Zeiten möglich.

Dynamische Zeiten werden für Echtzeitsimulationen verwendet, wenn die Physik-Simulation genau entsprechend der verstrichenen Zeit berechnet werden soll. Statische Zeiten sind dann angebracht, wenn Berechnungen entweder zu genau definierten, diskreten Zeitpunkten erfolgen sollen oder wenn ein Zeitraffer gewünscht ist. So führt beispielsweise ein sechshundertfacher Aufruf der Methode mit jeweils einer Sechzigstel-Sekunde als Parameter zu einem Fortschreiten der Simulation um zehn Sekunden. Je nach Leistungsfähigkeit des verwendeten Rechners und Komplexität der Simulation ist dies in Bruchteilen von Sekunden möglich.

Zu beachten ist ebenfalls, dass ausschließlich bei festen Zeiten wiederholbare Ergebnisse produziert werden können. Wenn ein dynamischer Aufruf mit immer unterschiedlichen Zeiten erfolgt, werden sich die Ergebnisse zweier ansonsten identischer Simulationsläufe voraussichtlich unterscheiden.

Während der gesamten Simulationsdauer können zu jedem Zeitpunkt über die *DefaultMotionStates* Informationen über die simulierten Körper abgerufen werden. So lässt sich entweder ein Protokoll der Simulation erstellen oder eine grafische Darstellung der Szene realisieren. Wenn der Informationsabruf zwischen zwei Simulationsschritten erfolgt, wird die seit dem letzten Simulationsschritt verstrichene Zeit bestimmt, um eine Interpolation zu ermöglichen.

Auch innerhalb eines Simulationslaufs dürfen verschiedene Körper sowie Constraints hinzugefügt oder entfernt werden. Dabei ist aber zu beachten, dass zu Körpern gehörende Constraints stets nach dem Einfügen der Körper hinzugefügt bzw. vor dem Löschen dieser gelöscht werden. Ein Constraint mit fehlendem Körper führt fast immer zu Instabilitäten der Simulation oder gar zu Programmabstürzen.

Außerdem sollte niemals von außen in die Simulation eingegriffen werden. Ein manueller Eingriff führt zu inkonsistenten Zuständen innerhalb der Physik-Engine die weitere (korrekte) Berechnungen unmöglich machen. Dies hat zur Folge, dass es zwar erlaubt ist, an Körper bestimmte Kräfte, Impulse oder Drehmomente anzulegen, aber auf keinen Fall manuell deren Position, Rotation oder innere Kräfte geändert werden sollten. Sobald ein Körper hinzugefügt wurde, darf er nicht mehr manuell bewegt, verschoben oder auf eine andere Art und Weise manipuliert werden. Eine Drehung oder Verschiebung darf ausschließlich über Anlegen einer passenden Kraft bzw. eines Drehmoments durch die Physik-Engine geschehen. Eine Änderung der *CollisionShape* ist ebenso wenig möglich wie das Ändern der Masse. Wenn solche Eigenschaften verändert werden sollen, muss der Körper zuerst entfernt und anschließend neu hinzugefügt werden. Anderenfalls führt der manuelle Eingriff entweder zum Absturz der Engine oder zu falschen und unzuverlässigen Ergebnissen.

**Einheiten in JBullet** Sämtliche Zahlenwerte werden in JBULLET einheitenlos als Gleitkommazahlen mit einfacher Genauigkeit<sup>19</sup> gespeichert.

---

<sup>19</sup>Es handelt sich um Zahlen des elementaren Datentyps *float*.

JBULLET nimmt implizit gewisse Einheiten an. So wird die Masse eines Körpers stets in Kilogramm interpretiert. Längeneinheiten werden in Metern gemessen, Kräfte in Newton usw. JBULLET funktioniert am besten, wenn sich Größen und Massen im ein- bis zweistelligen Bereich befinden. Bei größeren oder kleineren Objekten kommt es wegen der einfachen Genauigkeit schnell zu Rundungsfehlern. Daher kann es angebracht sein, die Physik-Welt vor der Simulation zu skalieren.<sup>20</sup> In dieser Arbeit war dies nicht nötig, da sich sowohl die Massen als auch die Ausmaße der Agenten in einem unkritischen Bereich befinden.

#### 3.1.2.4. Vereinfachungen

Wie nur unschwer an den vorhergehenden Ausführungen zu erkennen ist, sind eine Vielzahl an Befehlen und Klassen nötig, um überhaupt eine simple Simulation starten zu können. Da das EAS-Framework auch eine einfache Simulation von komplexen Umgebungen ermöglichen möchte, soll die Verwendung von JBULLET im Rahmen des EAS-Frameworks erleichtert werden. Das Einrichten einer Physik-Simulationsumgebung lässt sich in der Regel immer auf dieselbe Art und Weise bewerkstelligen. Deshalb bietet es sich an, dies in eigens dafür geschriebenen Klassen zu machen, so dass die Schritte von einem Benutzer des EAS-Frameworks nicht manuell durchgeführt werden müssen.

Aus diesem Grund wurden im Rahmen der vorliegenden Arbeit mehrere grundlegende Klassen entwickelt, die als Ausgangsbasis für weitere Projekte dienen können. Diese sollen hier kurz vorgestellt und erläutert werden.

**Eine abstrakte dreidimensionale Simulationsumgebung** Die erste wichtige Komponente einer jeden Simulation im EAS-Framework ist das Environment, sprich die Umgebung, in welcher die Agenten simuliert werden. Damit dreidimensionale Physik-Simulation überhaupt möglich werden, wurde die Klasse *AbstractEnvironment3D*<sup>21</sup> geschrieben. Diese stellt

---

<sup>20</sup>[http://www.bulletphysics.org/mediawiki-1.5.8/index.php?title=Scaling\\_The\\_World](http://www.bulletphysics.org/mediawiki-1.5.8/index.php?title=Scaling_The_World)

<sup>21</sup>`eas.simulation.spatial.sim3D.physicalSimulation.  
standardEnvironments.AbstractEnvironment3D`



als abstrakte Klasse sämtliche Grundfunktionen zur Verfügung, die nötig sind, um JBULLET im EAS-Framework nutzen zu können.

Dazu wird eine eigene Klasse erstellt, die von *AbstractEnvironment3D* erbt. In der erbenden Klasse muss der Super-Konstruktor aufgerufen werden, mit welchem dann eine Physik-Welt erzeugt wird, die sich für die Simulation von Festkörpern in einem dreidimensionalen Raum eignet. Sollte JBULLET in Zukunft CUDA<sup>22</sup> oder ähnliche Verfahren unterstützen, so müssten die verwendeten Algorithmen angepasst werden. Solange die Verwendung eines speziellen Verfahrens aber nicht gewünscht ist, kann von der Klasse ohne größere Anpassungen geerbt werden.

So kann auf einfache Weise ein Environment erzeugt werden, in welchem alle nötigen Einstellungen bereits vorgenommen wurden. Das selbst erstellte Environment verfügt dann über geerbte Methoden, um die Physik-Simulation ablaufen zu lassen. Es muss aus dem Scheduler lediglich die *step()*-Methode des Environments aufgerufen werden, um die Simulation in Sechzigstel-Sekunden-Schritten fortschreiten zu lassen. Die Frequenz von sechzig Hertz ist dabei voreingestellt und muss nicht mehr konfiguriert werden. Für abweichende Werte kann die Konfiguration geändert werden. Methoden zum Hinzufügen oder Löschen von Agenten sind ebenfalls implementiert.

**Ein abstrakter dreidimensionaler Agent** Neben der abstrakten Klasse für eine dreidimensionale Umgebung gibt es auch noch eine abstrakte Klasse für dreidimensionale Agenten. Die Klasse *AbstractAgent3D*<sup>23</sup> bietet alle Grundfunktionen und -eigenschaften, die nötig sind, um einen Agenten in einer JBULLET-Physik-Welt simulieren zu können. Über den Konstruktor, der von der erbenden Klasse aufgerufen werden muss, werden neben der eindeutigen Identifikationsnummer auch das Environment und die physikalischen Eigenschaften festgelegt. Sobald ein Agent über den Konstruktor er-

---

<sup>22</sup>CUDA steht für *Compute Unified Device Architecture*. Die Technik erlaubt es, Programme auf Grafikprozessoren auszuführen. Je nach Anwendungsfall sind bedeutende Leistungssteigerungen möglich.

<sup>23</sup>`eas.simulation.spatial.sim3D.physicalSimulation.  
standardAgents.AbstractAgent3D`

zeugt wird, wird dieser automatisch der JBULLET-Simulation hinzugefügt. Die Klasse bietet des Weiteren auch Methoden zur Positionsbestimmung der Agenten an, so dass der Umweg über das Auslesen des *DefaultMotionStates* nicht mehr nötig ist.

Zusammenfassend stellen diese beiden abstrakten Klassen den Grundstein für weitere Klassen dar, die für die physikalisch korrekte Simulation in einer dreidimensionalen Umgebung verwendet werden können. Auf die konkreten Klassen, welche für die im Rahmen dieser Arbeit durchgeführten Experimente verwendet wurden, soll in Abschnitt 3.3 der Arbeit eingegangen werden.

#### 3.1.2.5. Ein Video-Plugin zur grafischen Darstellung

Neben den Klassen für die eigentliche Simulation wurde auch ein Plugin für das EAS-Framework programmiert, welches die Visualisierung der JBULLET-Simulation erlaubt: das *VideoPlugin3D*.

Dem Quellcode von JBULLET waren diverse Beispiele beigelegt. Mehrere dieser Beispiele verwenden weitere Klassen, welche eine grafische Darstellung der Simulation durch die LIGHTWEIGHT JAVA GAME LIBRARY (LWJGL)<sup>24</sup> realisieren. Die LWJGL ist eine Bibliothek, die die Verwendung von OpenGL in Java ermöglichen möchte.

Das Ziel der Bibliothek ist es laut Erstellern nicht, das Programmieren von Spielen oder dreidimensionalen Szenen im Allgemeinen besonders einfach zu gestalten, sondern dies überhaupt erst in Java zu ermöglichen. Java bietet standardmäßig keine direkte Anbindung an OpenGL, so dass weitere Bibliotheken nötig werden, die diese Möglichkeiten bieten. Diese Lücke möchte die LWJGL füllen, indem sie alle benötigten Schnittstellen in einer einzigen Bibliothek anbietet.

Die bei JBULLET mitgelieferten Demo-Klassen erlauben eine dreidimensionale grafische Darstellung von Physik-Szenen, welche mit JBULLET erzeugt wurden. Dabei ist die Grafik sehr einfach gehalten, da die Klassen nur als Demo-Klassen zu verstehen sind. Trotzdem lässt sich für die Ex-

---

<sup>24</sup><http://lwjgl.org/>

perimente in dieser Arbeit eine ausreichende Qualität erzielen, welche alle nötigen Einzelheiten erkennen lässt. Darum wurden die Demo-Klassen als Ausgangsbasis für ein Plugin gewählt, um nicht eine komplette Grafik-Engine programmieren zu müssen.

Das daraus entstandene Plugin erzeugt während der Simulation eine dreidimensionale Ansicht der physikalischen Welt. Dazu wird von jedem Kollisions-Objekt, welches in `JBULLET` simuliert wird, der *DefaultMotionState* abgefragt. Dieser liefert (falls nötig interpolierte) Informationen über Rotation und Position des Kollisions-Objekts. Neben diesen beiden räumlichen Informationen wird außerdem noch die Form und Größe des Kollisions-Körpers abgefragt. Das *VideoPlugin3D* nimmt daraufhin eine perspektivische Verzerrung der korrekt positioniert und gedrehten Kollisions-Form vor, wodurch eine dreidimensionale Darstellung erzeugt wird.

Die Ansicht ist mittels Maus oder Tastatur dreh- und zoombar. So kann der Bildausschnitt für das jeweilige Experiment passend gewählt werden. Im Scheduler kann außerdem der zu beobachtende Bereich in der Welt oder auch der zu verfolgende Agent ausgewählt werden. Somit sind vielfältige Möglichkeiten einer grafischen Darstellung gegeben, um entweder Simulationen am Bildschirm verfolgen zu können oder Ergebnisse zu präsentieren.

In der aktuellen Version des Plugins ist die Grafik noch sehr einfach gehalten. So werden weder Texturen noch Schatten unterstützt. Die Beobachtung und Beurteilung von Agenten ist jedoch auch mit dieser einfachen grafischen Darstellung möglich. Ebenso können Bilder oder Videos der zu untersuchenden Wesen erzeugt werden.

Zu beachten ist jedoch, dass die `LWJGL` eigentlich keine Screenshots unterstützt, so dass zur Erzeugung eines Bildschirmbilds der komplette Grafikpuffer ausgelesen und umgewandelt werden muss. Anschließend wird dieser dann in einer Bilddatei gespeichert. Leider nimmt dieser Prozess relativ viel Zeit in Anspruch, so dass pro Screenshot teilweise bis zu einer Sekunde vergeht. Eine Erzeugung eines Videos mit 30 Bildern pro Sekunde wird so sehr schnell zu einem langwierigen Unterfangen. Der begrenzende Faktor ist hier interessanterweise nicht die Physik-Simulation oder die grafische Darstellung sondern das Umwandeln des Puffers in eine Bilddatei

bzw. in ein Video.

Wichtig ist weiterhin, dass im Scheduler, welcher die Videodatei erzeugen soll, keine dynamischen Zeitschritte zur Ansteuerung von JBULLET übergeben werden. Stattdessen sind konstante, diskrete Zeitschritte zu wählen, so dass die Bildwiederholfrequenz des Videos exakt mit den Zeitschritten übereinstimmt. Nur so wird das Video dann auch in Simulationsechtzeit aufgenommen und abgespielt.

## 3.2. Entwurf und Umsetzung

Bei dem Entwurf und der Entwicklung der Agenten, die für die Experimente verwendet werden, steht die Vermeidung zu großer Komplexität im Vordergrund. Erklärtes Ziel ist es, weder die Agenten noch die Genome zu komplex zu modellieren, sondern die gesamten Experimente dort, wo es möglich ist, sehr einfach zu gestalten.

Da das EAS-Framework zum Zeitpunkt der Diplomarbeit noch über keine dreidimensionalen Komponenten verfügte, galt es, alle nötigen Bausteine selbst zu entwickeln. Dazu zählen neben den eigentlichen Agenten mit ihren Steuerungseinheiten auch das Genom und die Fortpflanzung der Agenten. Die dazu angestellten Überlegungen und die Entwicklungen sollen in den folgenden Abschnitten beschrieben werden.

### 3.2.1. Der Agent

#### 3.2.1.1. Prototyp eines dreidimensionalen Agenten

Zunächst handelt es sich bei den hier entworfenen Agenten um recht einfache Wesen. Sie setzen sich immer aus Quadern – im Folgenden auch als Boxen bezeichnet – zusammen.

Die Boxen eines Agenten sind nummeriert, verfügen also jeweils über einen Index. Sie stehen insofern in einem Verhältnis zueinander, als dass jede Box genau einen Vorgänger hat, mit dem sie über ein einfaches Gelenk verbunden ist. Dieses Gelenk ist um eine Rotationsachse beweglich und kann vom Agenten gesteuert werden.

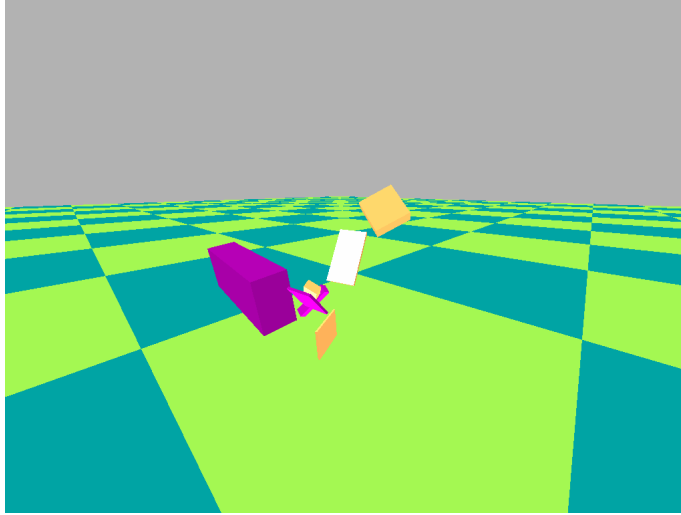


Abbildung 3.2.: Zufällig erzeugter Agent in der dreidimensionalen Simulationsumgebung

Eine besondere Rolle kommt der Kernbox zu. Diese besitzt den Index 0 und hat keinen Vorgänger, sie bildet sozusagen die Wurzel eines jeden Wesens.

Jede Box hat eine eigene Größe und Masse sowie eine relative Position zu ihrem Vorgänger. Die relative Position entfällt logischerweise bei der Kernbox, da diese keinen Vorgänger besitzt.

Neben den physischen Eigenschaften verfügt ein simulierter Agent auch über ein Gehirn, welches Steuerungsfunktionen übernehmen kann.

Der beschriebene Prototyp eines Agenten stellt eine sehr einfache und beschränkte Art eines dreidimensionalen Wesens dar. Trotzdem lässt sich auch mit diesen simplen Agenten die Evolution im dreidimensionalen Raum simulieren und beobachten. Abbildung 3.2 zeigt einen zufällig erzeugten Agent, wie er aus den angestellten Überlegungen hervorgegangen ist.

#### 3.2.1.2. Umsetzung des Prototyps

Auch wenn der Agent selbst relativ einfach aufgebaut ist, so sind doch einige Designfragen zu klären. Der oben beschriebene Prototyp wurde wie folgt umgesetzt.

**Boxen als kleinste Einheit** Zunächst erhält jede Box<sup>25</sup> eines Agenten eine eigene Indexnummer. Die Indexnummer 0 ist dabei für die Kernbox reserviert, alle weiteren Boxen werden aufsteigend nummeriert. Die Größe einer jeden Box wird in einem dreidimensionalen Vektor<sup>26</sup> gespeichert. Der Vektor ist zwar einheitenlos, JBULLET interpretiert die Komponenten jedoch als Längenangaben in Metern. Die Masse einer Box in Kilogramm entspricht exakt dem Volumen in Metern. Das Gewicht der Boxen ist somit proportional zu ihrem Volumen. Die Masse wurde so gering gewählt, damit die auch kleine Kräfte in den Gelenke ausreichen, um die Quader zu bewegen.

**Position der Boxen zueinander** Um die Position der Boxen zueinander festlegen zu können, wird ein weiterer dreidimensionaler Vektor<sup>27</sup> verwendet. Die einzelnen Komponenten dieses Vektors geben dabei die *relative* Position des Schwerpunkts der Vorgängerbox zum eigenen Schwerpunkt an. Da die Kernbox keine Vorgängerbox besitzt, entfällt bei ihr logischerweise der Positionsvektor.

Die relative Position aller anderen Boxen wird bei Erzeugung des Agenten stets so in Abhängigkeit der Quadergrößen skaliert, dass eine Box und ihr Vorgänger weit genug voneinander entfernt sind, um nicht direkt zu kollidieren. Damit sind die Positionsvektoren einheitenlos und geben nicht die Entfernung in Metern an. Stattdessen wird die Entfernung als ein Vielfaches der Seitenlängen der beteiligten Quader berechnet.

---

<sup>25</sup>Es wurde `com.bulletphysics.collision.shapes.BoxShape` als Kollisionsform verwendet. Andere geometrische Formen werden in der momentanen Version nicht verwendet, könnten jedoch implementiert werden.

<sup>26</sup>Dargestellt durch `javax.vecmath.Vector3f`, einem dreidimensionalen Vektor bestehend aus Zahlen vom Datentyp *float*.

<sup>27</sup>Auf dieselbe Art gespeichert wie der Größenvektor

Auf diese Weise kann die Position relativ zur Vorgängerbox genau festgelegt werden. Der Positionsvektor wird jeweils komponentenweise mit der Größe des Vorgängers bzw. der Box selbst multipliziert. Die beiden resultierenden Vektoren werden addiert, wodurch sich der Vektor ergibt, welcher die Verschiebung des Schwerpunkts der Vorgängerbox zum Schwerpunkt der eigentlichen Box darstellt.

Mit dieser Information lässt sich die Box relativ zum Vorgänger positionieren, ohne dass diese im Ruhezustand kollidieren könnten. Außerdem ist durch den eingestellten großen Abstand eine frühzeitige Kollision bei Bewegung unwahrscheinlich. Bei geringeren Abständen würde regelmäßig die Situation auftreten, dass sich zwei benachbarte Boxen verhaken und sich so in ihren Bewegungen einschränken.

**Verbindung der Boxen miteinander durch Gelenke** Eine Box ist jeweils mit ihrem direkten Vorgänger durch ein Gelenk verbunden, welches eine Rotation um eine der drei Raumachsen erlaubt. Die anderen zwei Rotationsachsen sowie die drei Translationsachsen bleiben gesperrt.

In der vorliegenden Implementierung wird dies durch ein generisches Gelenk gelöst, welches theoretisch sechs Freiheitsgrade ermöglicht, wobei aber die oben genannten gesperrt werden. Hierfür wurde eine geeignete bereits vorhandene Klasse<sup>28</sup> von JBULLET gewählt, da sie die meisten Konfigurationsmöglichkeiten zur Verfügung stellt. Außerdem ist sie die einzige Klasse, die sogenannte Rotationsmotoren<sup>29</sup> unterstützt, mit welchen eine Veränderung des Winkels des Gelenks erst möglich wird.

Bei der Erzeugung eines Gelenks sind die zwei zu verbindenden Körper, sowie die Ansatzpunkte des eigentlichen Gelenks als Parameter erforderlich. Bei den zwei zu verbindenden Körpern handelt es sich um die eigentliche Box sowie die Vorgängerbox. Die Gelenkansatzpunkte werden über die jeweiligen Körperschwerpunkte und die relative Position der Boxen zueinander bestimmt. Dadurch liegt das Gelenk stets auf der direkten Verbindungsstrecke der beiden Schwerpunkte. Die genaue Position auf die-

---

<sup>28</sup>com.bulletphysics.dynamics.constraintsolver.Generic6DofConstraint

<sup>29</sup>com.bulletphysics.dynamics.constraintsolver.RotationalLimitMotor

ser Strecke ist von der Größe der Boxen abhängig: je größer eine Box im Vergleich zur anderen ist, desto weiter ist das Gelenk von dieser Box weg.

Um die Ansteuerung des Gelenks möglichst einfach zu halten ist nur ein Freiheitsgrad freigeschaltet. So ist das Gelenk entweder um die x-, die y- oder die z-Achse frei beweglich und kann um annähernd  $170^\circ$  gedreht werden ( $85^\circ$  vom Nullpunkt in eine,  $85^\circ$  in die andere Richtung). Der Abstand der Schwerpunkte zum Gelenkpunkt bleibt dabei aber immer konstant, eine Translation ist nicht möglich. Auch ist keine gleichzeitige Drehung um mehr als eine Achse möglich. Ein Gelenk bleibt stets nur für eine Rotationsachse freigeschaltet. Dies ermöglicht eine leichte Steuerung mittels einer einzigen Gleitkommazahl als Winkelangabe. Da jede normale Box mit ihrem Vorgänger über ein Gelenk verbunden ist (die Kernbox hat keinen Vorgänger) verfügt ein Agent immer über  $n - 1$  Gelenke, wobei  $n$  die gesamte Anzahl an Boxen darstellt.

**Ansteuerung der Gelenke durch Rotationsmotoren** Ein Gelenk allein ermöglicht jedoch noch keine vom Agenten ausgehende Winkeländerung. Es sorgt lediglich dafür, dass die Boxen ihre Position relativ zueinander einhalten, sich nicht verschieben und dass alle Rotationswinkel ohne aktive Ansteuerung nahe bei null sind. Dabei sind kleine Änderungen durch äußere Einflüsse zwar stets möglich, werden jedoch schnell von der Physik-Engine korrigiert. Die Gelenke sind nicht komplett starr sondern bleiben leicht elastisch. Dies ist nötig, um die Simulation stabil zu halten und Fehler durch Rundungen zu vermeiden. Wären die Gelenke komplett starr, würde die Engine enorme Kräfte simulieren, um die Constraints unverzüglich durchzusetzen, was zu einer instabilen Simulation führen kann.

Damit eine gesteuerte Winkeländerung möglich wird, müssen, wie oben bereits angedeutet, Rotationsmotoren eingesetzt werden. JBULLET stellt solche Motoren zur Verfügung.<sup>30</sup> Diese können vom Agenten als sogenannte Aktoren verwendet werden. Jedes Gelenk verfügt über einen eigenen Motor, der den gewünschten Winkel am Gelenk einstellt. Zu Beginn der

---

<sup>30</sup>`com.bulletphysics.dynamics.constraintsolver.RotationalLimitMotor`



Simulation sind alle Winkel stets auf  $0^\circ$  festgelegt, im Verlauf der Simulation können diese dann vom Künstlichen Neuronalen Netz des Agenten verändert werden.

Dazu wird die gewünschte *Winkeländerung* als Gleitkommazahl vom Künstlichen Neuronalen Netz ausgegeben. Die Ausgabe des Künstlichen Neuronalen Netzes ist normiert und kann einen Wert von  $-1,0$  bis  $1,0$  annehmen.

Die eigentliche Winkeländerung wird dann wie folgt durchgeführt. Der zuvor eingestellte Winkel wird gemessen und normiert. Ein Winkel von  $-85^\circ$  entspricht dem Wert  $-1,0$ .  $0^\circ$  entsprechen dem Wert  $0,0$  und  $+85^\circ$  entsprechen dem Wert  $1,0$ . Mögliche Zwischenwerte werden proportional dazu berechnet. Der so bestimmte Wert wird mit  $\pi$  multipliziert. Die Ausgabe des Künstlichen Neuronalen Netzes wird ebenfalls mit  $\pi$  multipliziert. Daraufhin werden beide Ergebnisse addiert und der Sinus der Summe bestimmt. So ergibt sich wieder ein Wert zwischen  $-1,0$  und  $1,0$ , was wiederum dem gewünschten Winkel von  $-85^\circ$  bis  $+85^\circ$  entspricht.

$$\text{normierterWinkel}_{\text{neu}} = \sin(\text{normierterWinkel}_{\text{alt}} * \pi + \text{normierteWinkeländerung} * \pi)$$

Die gewählte Methode hat den großen Vorteil, dass dem Motor unabhängig von der Ausgabe des Gehirns stets gültige Werte übergeben werden. Würde die gewünschte Änderung direkt auf den vorherigen Wert addiert werden, entstünden schnell Werte, deren Betrag größer als  $1,0$  ist, womit ungültige Winkel gefordert würden. Ein direktes Angeben des gewünschten Winkels (und nicht der Winkeländerung) hingegen hat in mehreren Probesimulationen dazu geführt, dass Agenten entstehen, welche zu einem bestimmten Zustand konvergieren, in welchem sie dann verharren ohne sich zu bewegen. Um dies zu umgehen wurde der oben vorgestellte Ansatz gewählt, der selbst dann zu einer Winkeländerung führt, wenn immer derselbe Wert vom Künstlichen Neuronalen Netz ausgegeben wird (abgesehen vom Wert  $0$ , was keiner Änderung entspräche).

Die Bewegung der Gelenke wurde auf  $170^\circ$  ( $\pm 85^\circ$ ) beschränkt, da die JBULLET-Implementierung der Gelenke eine Veränderung von maximal

90° in jede Richtung erlaubt. Sobald diese jedoch annähernd erreicht werden, treten extrem hohe Drehmomente auf, welche von der Physik-Engine eingesetzt werden, um die Gelenkgrenzen durchzusetzen. Dies wiederum führt zu einem unkontrollierten „Zappeln“ und Zurückschlagen der Boxen, da die Gelenke wegen der hohen Kräfte durchschlagen und so die andere Gelenkgrenze erreichen. In diversen Probesimulationen führte dies zu Populationen, die ausschließlich aus fehlerhaften Agenten bestanden. Diese zeigten statt einer kontrollierten Fortbewegung nur oben genanntes „Zappeln“. Das Herabsetzen des Bereichs von 180° ( $\pm 90^\circ$ ) auf 170° ( $\pm 85^\circ$ ) lässt normalerweise genügend Spielraum für Änderungen durch äußere Einflüsse (Aufschlagen auf dem Boden, Zusammenstoß mit anderen Boxen usw.), ohne dass der kritische Bereich von mehr als 90° bzw. weniger als -90° erreicht wird.

**Messung der Gelenkwinkel** Neben den eben vorgestellten Aktoren gibt es auch noch Sensoren, welche von Agenten bzw. deren Künstlichen Neuralen Netzen zur Steuerung benutzt werden. Die wichtigsten Sensoren, die für jedes Gelenk vorhanden sind, sind die Winkelsensoren. Wie zu Beginn dieses Abschnitts erklärt, sind eine Box und ihr Vorgänger stets mit einem Gelenk verbunden. Für jedes Gelenk gibt es einen Aktor – den soeben vorgestellten Rotationsmotor – und einen Sensor – den Winkelsensor.

Der Sensor bestimmt den Winkel direkt über die Physik-Engine. Dazu wird aus dem zu dem Gelenk gehörigen Constraint der Rotationswinkel der freigeschalteten Achse ausgelesen und normiert. Auch hier entspricht wie bei den Aktoren bzw. Rotationsmotoren der Wert -1,0 einem Winkel von -85°, der Wert 0,0 einem Winkel von 0° und der Wert 1,0 einem Winkel von +85°. Zwischenwerte werden analog zu oben ebenfalls wieder proportional berechnet.

Die Winkel der gesperrten Achsen können sich zwar minimal ändern, werden jedoch nicht gemessen, da der Spielraum weniger als 2° beträgt und der Winkel nicht gesteuert werden kann. Der Spielraum ist lediglich vorhanden, um eine stabile Simulation zu ermöglichen, da komplett fixe Winkel aufgrund von Rundungsfehler zu enormen Korrekturkräften durch

die Physik-Engine führen können, die wiederum außer Kontrolle geratene Simulationen zur Folge haben, welche keine zuverlässigen Ergebnisse liefern können.

**Bestimmung der Bewegung und Lage** Weitere Sensoren, die optional hinzugefügt werden können sind Bewegungs- und Lagesensoren. Der Bewegungssensor bestimmt die Veränderung des Schwerpunkts der Kernbox eines Agenten im Vergleich zum vorhergehenden Berechnungsschritt. Diese Veränderung wird normiert, so dass alle Komponenten des Vektors zwischen  $-1,0$  und  $1,0$  liegen. Damit lässt sich die Bewegungsrichtung des Agenten vom letzten Punkt zum aktuellen darstellen.

Der Lagesensor stellt die Lage der Kernbox im Raum als dreidimensionalen Vektor dar. Auch hier werden die Komponenten normiert.

**Das Gehirn des Agenten** Damit ein Agent die Sensoren und Aktoren auch verwenden kann, um so seinen Körper bewegen zu können, muss er über ein Gehirn verfügen. Dieses muss Eingänge besitzen, welche gemessene Werte der Umgebung entgegen nehmen können und Ausgänge, welche Werte an Aktoren bzw. die Umwelt weitergeben können. Darüber hinaus muss es eine Steuerungsmöglichkeit besitzen, die es ihm möglich macht, zu bestimmten Eingabewerte passende Ausgabewerte zu berechnen.

Da die gezielte Fortbewegung im dreidimensionalen Raum eine sehr komplexe Aufgabe darstellt, wurden Künstliche Neuronale Netze als Basis des Gehirns gewählt. Die Aufgaben des Gehirn des Wesens umfassen das Auslesen der Sensoren, das Berechnen der Ausgabewerte und das anschließende Weitergeben dieser an die Aktoren. Das Künstliche Neuronale Netz dient dabei als zentrale Berechnungseinheit.

Die Beschreibung der Künstlichen Neuronalen Netze und ihrer Funktionsweise erfolgt genauer in Abschnitt 2.3.3.

#### 3.2.2. Das Genom

Alle Informationen, die zur Beschreibung und Erzeugung eines Agenten nötig sind, werden in einem Genom gespeichert. So lässt sich aus dem Genotyp (dem Genom in *String*-Repräsentation) der Phänotyp (der eigentliche Agent mit seinem Verhalten) erzeugen. Das Genom wurde im Hinblick auf mehrere Ziele entwickelt:

- Alle Informationen sollen einfach und kompakt gespeichert werden.
- Das Genom soll menschenlesbar sein.
- Das Genom muss diverse Mutationsoperatoren ermöglichen.

##### 3.2.2.1. Grundlegende Eigenschaften

Das Ziel, das Genom möglichst einfach und kompakt zu halten, führt natürlich dazu, dass gewisse Entscheidungen schon im Vorfeld getroffen werden müssen und diese so durch das Genom nicht mehr beeinflusst werden können. Dadurch wird zwar einerseits verhindert, dass die Genome zu komplex werden, andererseits werden natürlich auch Einschränkungen vorgenommen. So ist beispielsweise in den hier vorgestellten Experimenten aus Gründen der Einfachheit die einzige mögliche geometrische Form eine Box. Damit ist es nicht nötig, ein Gen für die geometrische Form im Genom vorzusehen, was ein kürzeres und einfacheres Genom ermöglicht. Die Wesen sind aber auf aus Quadern aufgebaute Körper beschränkt.

Um unnötige Verkomplizierungen zu vermeiden wurde das Genom so aufgebaut, dass die Informationen über die einzelnen Boxen nach aufsteigendem Index aneinander gehängt werden. Im Anschluss an diese Informationen kommt die Beschreibung des Künstlichen Neuronalen Netzes. So lässt sich jeder Baustein des Agenten einzeln und der Reihenfolge nach betrachten.

Das Genom wird als lesbare Zeichenkette gespeichert.<sup>31</sup> So ist sichergestellt, dass das Genom menschenlesbar ist. Wenn das Genom als *Object*

---

<sup>31</sup>Es handelt sich um eine Zeichenkette vom Typ *String*, welche in einer Textdatei gespeichert werden kann.

*Stream* serialisiert würde, wäre dies nicht mehr der Fall. Daher wurde dieser Weg ausgeschlossen, auch wenn das Speichern und Abrufen der Genome so leichter gewesen wäre.

Alle Komponenten sind im Genom getrennt voneinander gespeichert und können einzeln manipuliert werden. Dies ermöglicht eine Vielzahl von Mutationsoperatoren auf dem Genom, welche in Abschnitt 3.2.3.1 vorgestellt werden sollen.

#### 3.2.2.2. Aufbau des Genoms

Anhand eines zufällig erzeugten Genoms werden im Folgenden der Aufbau und die Eigenschaften der für diese Diplomarbeit entwickelten Genome erläutert. Das hier beispielhaft betrachtete Genom sieht wie folgt aus.<sup>32</sup>

```
(0.58606833, 1.8095354, 1.6343048):/:
1:0:(1.950554, -0.41319042, -0.7081524):(0.61797065, 0.6198123, 1.3658949):0:/:
2:0:(-0.9386499, -0.30109313, 0.38287073):(0.69393146, 0.68010587, 1.0514721):0:/:
3:0:(-1.1893494, 1.0382562, 1.0160137):(1.2136787, 1.1530135, 2.3453896):2:
///:
Inputs: 3XXXOutputs: 3;
N00:0.0I0.0I0.0I0.0I0.0I0.0;
N01:0.0I0.0I0.0I0.0I0.0I0.0;
N02:0.0I0.0I0.0I0.0I0.0I0.0;
N03:0.0I0.0I0.0I0.0I0.0I0.0;
N04:0.0I0.0I0.0I0.0I0.0I-1.5106384953089418I0.0;
N05:0.0I0.0I0.0I0.0I0.0I0.0;
N06:0.0I0.0I0.0I0.0I0.0I-0.5955218687753316;
#
```

Das Genom soll sowohl eine leichte Verarbeitung in den Experimenten ermöglichen, als auch die Interpretation durch Menschen gestatten. Deshalb sind alle Informationen im Klartext im Genom gespeichert. Eine Verschlüsselung oder Komprimierung erfolgt nicht. Um jedoch der erleichterten Verarbeitung Rechnung zu tragen, wurden sämtliche Informationseinheiten im Körper durch einen Doppelpunkt (:) und im Künstlichen Neuronalen Netz durch ein Semikolon (;) getrennt. Diese Zeichen dienen einzig und allein dem *Parser*, um das Einlesen und Verarbeiten der Genome zu vereinfachen.

<sup>32</sup>Durch die Verwendung von Zeilenumbrüchen wurde die Lesbarkeit des Genoms verbessert. Üblicherweise wird das gesamte Genom in einer einzigen Zeile abgespeichert.

**Physischer Aufbau des Agenten** Ein Genom besteht immer aus zwei Hauptteilen. Im ersten Teil (vor der Zeichenkette `///`) sind alle Informationen über den physischen Aufbau des Agenten gespeichert.

```
(0.58606833, 1.8095354, 1.6343048):/:  
1:0:(1.950554, -0.41319042, -0.7081524):(0.61797065, 0.6198123, 1.3658949):0:/:  
2:0:(-0.9386499, -0.30109313, 0.38287073):(0.69393146, 0.68010587, 1.0514721):0:/:  
3:0:(-1.1893494, 1.0382562, 1.0160137):(1.2136787, 1.1530135, 2.3453896):2:
```

Wie bereits zuvor erklärt, dienen die Doppelpunkte lediglich der Vereinfachung des Einlesevorgangs und der Handhabung der Genome durch das EAS-Framework. Der Schrägstrich (`/`) trennt die Informationen zweier Boxen voneinander ab.

Die erste Box ist stets die Kernbox. Da diese immer den Index 0 hat und keinen Vorgänger besitzt, muss lediglich ihre Größe gespeichert werden. Die Größe einer Box wird im Genom mit einem dreidimensionalen Vektor gespeichert. Der Vektor besteht aus drei Zahlen vom Typ *float*, die jeweils durch ein Komma getrennt eingeklammert sind.

```
(0.58606833, 1.8095354, 1.6343048)
```

Außer der Kernbox gibt es in einem Agenten immer mindestens eine weitere Box, im vorliegenden Beispiel sogar drei weitere. Die Boxen werden alle nach demselben Schema beschrieben.

```
1:0:(1.950554, -0.41319042, -0.7081524):(0.61797065, 0.6198123, 1.3658949):0:/:
```

Die erste Zahl gibt den eigenen Index der Box an. Dieser wird verwendet, um eine Box eindeutig identifizieren zu können, um z.B. im Rahmen von Mutationen Eigenschaften zu ändern.

Nach der eigenen Indexnummer kommt der Index der Vorgängerbox. Mit dieser Zahl wird der Vorgänger des Quaders festgelegt. Im Anschluss an die Indexnummer des Vorgängers findet sich im Genom die Position auf dem Vorgänger. Die Position auf dem Vorgänger wird ebenso wie die Größe als dreidimensionaler Vektor gespeichert. Auch hier sind die einzelnen Komponenten Gleitkommazahlen vom Typ *float*. Jede der drei Komponenten gibt dabei die relative Lage der Massenschwerpunkte der Boxen zueinander auf einer Achse an.

Auf die Position des Vorgängers folgt ein weiterer Vektor. Dieser gibt die Größe der Box an. Die letzte Information, die in einer Box vorhanden ist, wird wieder durch eine einzelne Zahl kodiert. Mit dieser Zahl wird die freie Rotationsachse gespeichert. Eine 0 bedeutet hierbei, dass Rotationen um die x-Achse der Verbindung der Box mit ihrem Vorgänger gestattet sind. Eine 1 ermöglicht die Steuerung der y-Achse und eine 2 die Steuerung der z-Achse. Sobald alle Informationen über den physikalischen Aufbau im Genom gespeichert wurden, wird dieser Teil mit der Zeichenkette `///` abgeschlossen.

**Aufbau des Künstlichen Neuronalen Netzes** Der zweite Teil enthält das gesamte Künstliche Neuronale Netz mit allen Verbindungen und Gewichten:

```
Inputs: 3XXXOutputs: 3;
N00:0.010.010.010.010.010.010.0;
N01:0.010.010.010.010.010.010.0;
N02:0.010.010.010.010.010.010.0;
N03:0.010.010.010.010.010.010.0;
N04:0.010.010.010.010.01-1.510638495308941810.0;
N05:0.010.010.010.010.010.010.0;
N06:0.010.010.010.010.010.01-0.5955218687753316;
```

Zu Beginn werden hier die Anzahl der Eingabe- sowie Ausgabeneuronen festgelegt. Diese speziellen Neuronen sind nötig, um die Sensorwerte einzulesen bzw. die Aktorwerte schreiben zu können. Alle anderen Neuronen befinden sich in den sogenannten *hidden layers* und können weder Sensoren noch Aktoren direkt ansprechen. Da die Eingabe- und Ausgabeneuronen diese Funktion als einzige übernehmen können, sind sie geschützt und können bei Mutationen nicht gelöscht werden.

Nach diesen Informationen folgt die Beschreibung des Künstlichen Neuronalen Netzes in Form einer Matrixdarstellung. Die einzelnen Zeilen dieser Matrix sind mit  $N_{xx}$  beschriftet, wobei  $xx$  für die zweistellige Zeilennummer steht. Das Ende einer Zeile wird durch ein Semikolon markiert, während die Spalten mit dem Zeichen  $I$  voneinander abgegrenzt werden. Auf die Bedeutung der Matrix wird in Abschnitt 2.3.3 genauer eingegangen.

An dieser Stelle sei nur gesagt, dass die Werte in der Matrix die Synapsen zwischen zwei Neuronen sowie deren Gewichte darstellen.

Ein jedes Genom wird mit einer Raute (#) abgeschlossen.

#### 3.2.2.3. Einordnung des Genoms

Mit dem für diese Diplomarbeit entwickelten Genom lassen sich potentiell unendlich viele Agenten beschreiben. Dennoch bleibt das Genom kompakt und einfach aufgebaut. Des Weiteren bleibt die textuelle Darstellung für Menschen lesbar. Neben den körperlichen Eigenschaften wird ebenfalls das gesamte Künstliche Neuronale Netz abgespeichert. Somit lässt sich sowohl die physische Struktur als auch die gesamte Steuerungseinheit eines jeden Agenten abspeichern und laden.

Während bei dem vorliegenden Genom nur eine recht einfache Graphenstruktur erkennbar ist, sind in der Literatur häufig komplexere, aufwendigere Graphen zu finden. SIMS [Sim94a, Sim94b] verwendet beispielsweise Graphen, die eine Rekursion ebenso wie Symmetrien erlauben. Genome mit Graphenstruktur sind u.a. auch in [Krč08] zu finden. Diese Art von Genom ermöglicht zwar einerseits die Beschreibung von fraktalen Wesen, andererseits steigt die Komplexität deutlich an. Um die gesamten Experimente so einfach wie möglich zu halten, wurde das hier entwickelte Genom anstelle komplexer durch Graphen dargestellten Genome verwendet. Es muss jedoch erkannt werden, dass das verwendete einfachere Genom auch eine Einschränkung darstellen kann. [Sta06]

#### 3.2.3. Fortpflanzung

Jede Generation von Agenten wird evaluiert. Am Ende dieser Evaluation wird eine gewisse Zahl an Agenten ausgewählt, die ihr Erbgut, ihr Genom, weitergeben dürfen. Diese Auswahl kann zufällig erfolgen, auf dem Fitnesswert basieren oder einem anderen, komplexeren Verfahren gehorchen. Auf welche Weise die Auswahl auch erfolgen mag, im Anschluss gibt es mehrere Möglichkeiten, wie ein Agent sein Erbgut weitergeben kann.



### 3.2.3.1. Mutationen

Das Kopieren des Erbguts ermöglicht die Fortpflanzung eines Agenten ohne einen zweiten Elternteil. Um nicht erneut exakt dasselbe Wesen zu erhalten, wird auf die Mutation zurückgegriffen.

Sobald ein Agent ausgewählt wird, wird dessen Erbgut dupliziert. Das kopierte Erbgut wird schließlich an einer oder mehreren Stellen zufällig modifiziert bzw. mutiert. Das dadurch entstandene Genom ist eine Mutation des vorhergehenden Genoms.

Für die durchgeführten Experimente wurden diverse Mutationsoperatoren entwickelt und implementiert. Diese Veränderungen können bei den hier verwendeten Agenten entweder den Körper oder das Künstliche Neuronale Netz betreffen. Dabei wurden eine Vielzahl von verschiedenen Mutationen implementiert. Zunächst sollen die Mutationen, die den Körper des Agenten betreffen, vorgestellt werden, anschließend jene, die auf dem Künstlichen Neuronalen Netz operieren.

**Mutation des Körpers** Da ein Agent aus mehreren Quadern unterschiedlicher Größe besteht, liegt es nahe, die Größe eines solchen Quaders zu verändern. Es wird zufällig eine Box des Wesens ausgewählt und deren ursprüngliche Größe ausgelesen. Diese Größe wird durch Addition eines zufälligen Vektors verändert, wobei jedoch stets gewährleistet ist, dass die Box ein positives Volumen behält.

Es bietet sich ebenfalls an, neben der Größe eines Quaders dessen Position auf dem Vorgänger zu ändern. Dieser Operator entfällt logischerweise für den Kern eines Agenten, da der Kern keinen Vorgänger besitzt. Für alle anderen Boxen erfolgt die Mutation wieder durch Veränderung des Vektors, der die Position auf dem Vorgänger beschreibt.

Eine Box ist stets mit ihrem Vorgänger verbunden. Die Verbindung erfolgt durch ein Gelenk, welches nur um eine Achse gedreht werden kann. Ein weiterer Mutationsoperator erlaubt es, diese freigeschaltete Achse zu verändern. Dabei bleiben jedoch auch weiterhin nur Rotationen um eine einzige Achse gestattet. Das Gelenk bleibt für alle weiteren Rotations-

wie Translationsachsen gesperrt.

Es können jedoch nicht nur die Eigenschaften einer Box verändert werden. Weiterhin ist es auch möglich, ganze Boxen hinzuzufügen. Die dabei festzulegenden Eigenschaften werden alle im Mutationsprozess zufällig bestimmt. Dazu gehören neben der Größe der Box auch deren Vorgänger, die Position auf dem Vorgänger und die freigeschaltete Achse.

Sobald dem Agenten eine weitere Box hinzugefügt wird, entsteht automatisch ein neues Gelenk im Agenten. Dieses verbindet den neuen Quader mit seinem Vorgänger. Damit das neu entstandene Gelenk auch vom Agenten verwendet werden kann, muss das Künstliche Neuronale Netz angepasst werden. Die Anpassung erfolgt automatisch durch den Mutationsoperator, so dass die neue Box direkt gesteuert werden könnte. Um dies zu erreichen wird dem Künstlichen Neuronalen Netz ein neues Eingabeneuron hinzugefügt, welches die Sensorwerte, die das Gelenk liefert, auslesen kann. Des Weiteren kommt ein neues Ausgabeneuron hinzu, mit welchem der Aktor mit Werten versorgt werden kann.

Um ein stetiges Größerwerden und Wachsen der Agenten zu verhindern, können im Rahmen einer Mutation auch Boxen aus dem Agenten entfernt werden. Dazu wird die zu entfernende Box aus dem Genom gelöscht und alle an diesem Quader befindlichen Nachfolger werden an den Vorgänger der entfernten Box angehängt. Der Kern bleibt jedoch stets erhalten und kann nicht gelöscht werden. Auch bei diesem Mutationsoperator ist es nötig, das Künstliche Neuronale Netz des Wesens anzupassen. Sobald eine Box entfernt wird, verschwindet mit ihr auch ein Gelenk und damit auch der zugehörige Winkelsensor und der Stellmotor. Das Eingabeneuron für den Sensor wird ebenso gelöscht wie das Ausgabeneuron für den Aktor.

Das Verschieben einer Box von einem Vorgänger auf einen anderen wurde nicht explizit als Mutationsoperator formuliert. Dies kann aber beispielsweise dann zufällig geschehen, wenn eine Box gelöscht wird und eine andere mit ähnlichen Eigenschaften an einer anderen Stelle im Agenten wieder hinzugefügt wird.

**Mutationen des Künstlichen Neuronalen Netzes** Neben den vorgestellten Mutationsoperatoren, die auf dem Körper arbeiten, gibt es weitere, welche nur das Künstliche Neuronale Netz verändern.

Um komplexere Steuerungen durchführen zu können, benötigt das Künstliche Neuronale Netz neben den Eingabe- und Ausgabeneuronen weitere, sogenannte innere Neuronen. Einer der Mutationsoperatoren fügt deshalb ein inneres Neuron hinzu, welches dem Künstlichen Neuronalen Netz möglicherweise komplexere Berechnungen erlaubt. Es können bei Mutationen im Künstlichen Neuronalen Netz nur innere Neuronen hinzugefügt werden. Eingabe- bzw. Ausgabeneuronen werden immer nur dann hinzugefügt, wenn eine Box zum Agenten hinzukommt und damit ein neuer Winkelsensor zur Verfügung steht.

Wie schon beim Körper so ist es auch beim Künstlichen Neuronalen Netz nicht erwünscht, dass dieses unbegrenzt wächst. Aus diesem Grund besteht auch hier die Möglichkeit, dass Neuronen entfernt werden. Wie beim Hinzufügen gilt auch hier, dass ausschließlich innere Neuronen beeinflusst werden. So können keine Eingabe- oder Ausgabeneuronen gelöscht werden, ohne dass die zugehörige Box gelöscht worden wäre. Es ist nicht sinnvoll, das Löschen von Eingabe- oder Ausgabeneuronen zu gestatten, da damit die Steuerung eines Gelenks komplett wegfallen würde. Die (vorübergehende) Nichtnutzung einer Box oder eines Gelenks ist weiterhin möglich, indem die entsprechenden Synapsen im Gehirn entfernt werden, was zur Folge hat, dass der Aktor keine Veränderungen durchführt. Beim Löschen eines Neurons werden alle möglicherweise vorhandenen Verbindungen zu anderen Neuronen ebenfalls entfernt, so dass diese nicht „ins Leere“ führen.

Es können jedoch nicht nur gesamte Neuronen hinzugefügt bzw. mit allen Verbindungen gelöscht werden. Es ist stattdessen auch möglich, einzelne Verbindungen zwischen zwei Neuronen herzustellen. Beim Erzeugen eines Künstlichen Neuronalen Netzes sind die Neuronen nicht miteinander verbunden. Dies geschieht erst im Rahmen von Mutationen. Der besagte Mutationsoperator wählt zufällig zwei Neuronen aus und verbindet diese mit einer neuen, zufällig erzeugten Synapse.

Verbindungen können jedoch nicht nur erzeugt sondern auch gelöscht

werden. Beim Löschen einer Verbindung zwischen zwei Neuronen verbleiben die Neuronen selbst weiterhin im Künstlichen Neuronalen Netz. Es wird lediglich die Synapse aufgelöst, so dass die beteiligten Neuronen getrennt werden. Die Verbindungen zu anderen Neuronen bleiben weiterhin bestehen.

Zuletzt ist es auch noch möglich, bestehende Synapsen zwischen den Neuronen zu verändern. Die Veränderung bezieht sich dabei auf die bestehenden Verbindungen und deren Gewichte. Es werden die Synapsengewichte der verbundenen Neuronen durch Addition zufälliger Werte verändert. Die Verbindungen bestehen aber weiterhin zwischen denselben Neuronen, so dass nach der Mutation noch dieselben Neuronen miteinander verbunden sind. Das „Umbiegen“ von Verbindungen von einem Neuron auf ein anderes ist nicht implementiert, da es dem Löschen einer Verbindung und dem Hinzufügen einer anderen entspricht.

**Anmerkungen zur Zufälligkeit der Mutationen** Alle vorgestellten Mutationen – sowohl auf dem Körper als auch auf dem Künstlichen Neuronalen Netz – erfolgen genau genommen nur pseudozufällig. Dazu wird den Mutationsoperatoren ein Pseudozufallszahlengenerator übergeben, der zu Beginn der Simulation mit einem bestimmten *random seed* gestartet wurde. Dies ist erwünscht, da so zwei Simulationsläufe mit demselben Startwert exakt gleich ablaufen. So werden Simulationsläufe nachvollziehbar und wiederholbar. Die Mutationen können trotzdem als zufällig angesehen werden, da das Initialisieren des Pseudozufallszahlengenerators mit verschiedenen Parametern möglich ist. [Col11]

Vom Benutzer sind im Vorfeld der Simulation diverse Mutationsgewichte festzulegen. Diese geben zuerst das Verhältnis von Mutationen auf der Körper und auf dem Künstlichen Neuronalen Netz an. Des Weiteren gibt es Parameter, die die Verhältnisse der einzelnen Mutationsoperatoren untereinander steuern. So ist es durch geeignete Parameterwahl möglich, ausschließlich Mutationen des Künstlichen Neuronalen Netzes zu untersuchen oder nur Mutationen der Boxgrößen zu erlauben usw.

#### 3.2.3.2. Crossover

Bei der Entwicklung der Agenten und der Experimente wurde im Vorfeld auch die Möglichkeit eines sogenannten Crossovers zwischen zwei Agenten in Erwägung gezogen. Im Gegensatz zur Mutation geht dabei nicht ein Agent aus einem einzigen Elternagenten hervor. Stattdessen werden die Eigenschaften zweier Agenten miteinander kombiniert, um einen Nachfolger zu erzeugen. Dabei wird das Ziel verfolgt, die guten Eigenschaften zweier Elternagenten zu kombinieren, so dass ein Nachfolger entsteht, der die Vorzüge seiner Vorfahren vereint und eine höhere Fitness erreichen kann.

Bei der Ausarbeitung der Agenten und der Experimente wurde die Möglichkeit der Kreuzung zweier Individuen jedoch ausgeschlossen. Zwei zufällig erzeugte Agenten unterscheiden sich in der Regel signifikant. Neben der Anzahl der Boxen, der Position dieser zueinander und der Gelenkart sind fast immer auch die Künstlichen Neuronalen Netze gänzlich verschieden. Es ist nicht zu erwarten, dass zwei Agenten, die sich vollkommen unabhängig voneinander entwickelt haben, gemeinsam einen Nachfahren erzeugen können, der eine ähnlich hohe Fitness hat.

Selbst wenn beide Agenten durch eine Reihe von Mutationen jeweils eine hohe Fitness erlangt haben, so spielt die Kompatibilität dennoch eine wichtige Rolle. Durch die vorhergehenden Mutationen verändert sich der Agent sowohl in Bezug auf das Aussehen als auch in Bezug auf das Künstliche Neuronale Netz. So entstehen einzigartige Individuen, deren Künstliches Neuronales Netz sich speziell auf den entstandenen Körper angepasst hat. Wenn zwei solche – vollkommen unterschiedliche – Individuen miteinander kombiniert werden, entsteht ein Wesen, welches für gewöhnlich nicht bewegungsfähig ist.

Das beschriebene Phänomen konnte in testweise durchgeführten Versuchen häufig beobachtet werden. Dazu wurden die Genome zweier Agenten mit relativ hoher Fitness ausgelesen und zufällig oder manuell miteinander kombiniert. In allen Fällen entstanden Individuen, die keine ausgeprägte Bewegungsfähigkeit zeigten. Entweder kam es zu Problemen beim physischen Aufbau der Agenten oder der Steuerung der Gelenke.

Oft zu beobachten waren sich gegenseitig blockierenden Boxen oder auch Quader, die ineinander ragten. Diese Agenten wären selbst mit einer passenden Steuerung nicht in der Lage gewesen, eine geregelte Bewegung durchzuführen. Sobald sich die Boxen so stark einschränken, dass eine Winkeländerung kaum oder gar nicht möglich ist, kann auch ein eigentlich funktionierendes Künstliches Neuronales Netz keinerlei Bewegung ermöglichen.

Gleiches gilt für Boxen, die ineinander ragen. Durch die im Genom vorgeschriebenen Positionen auf dem Vorgänger kann es vorkommen, dass zwei oder mehrere Boxen so im Agenten positioniert werden, dass sie sich durchdringen. Da es sich bei den Boxen aus Sicht der Physik-Engine aber um ausgefüllte, massive Kollisionskörper handelt, ist ein Durchdringen nicht gestattet. Dies führt dazu, dass die Physik-Engine große Kräfte auf die Boxen wirken lässt, um diese wieder auseinander zu drücken. Diesen Kräften werden jedoch wiederum Kräfte und Drehmomente der Gelenke entgegengesetzt. Die vom Künstlichen Neuronalen Netz angegebenen Winkeländerungen können dann nicht durchgesetzt werden. Vielmehr kommt es zu einem Aufschaukeln von entgegengesetzten Kräften. Die Physik-Engine ist bestrebt, gleichzeitig die Kollisionen aufzulösen und die Constraints durchzusetzen. Das Aufschaukeln führt zu immensen Kräften, die wiederum dafür sorgen, dass die gesamte Simulation instabil wird. Die Folge sind hin- und herschlagende Gelenke und ein unkontrolliertes Zucken der Agenten, ohne dass das Künstliche Neuronale Netz des Wesens die Steuerung noch übernehmen könnte.

Ein ähnliches Verhalten der Agenten ließ sich aber auch bei Wesen beobachten, die rein vom physischen Aufbau bewegungsfähig sein könnten. Obwohl die Elternteile offensichtlich bezüglich des Körperaufbaus kompatibel waren, konnte keine erfolgreiche Bewegung beobachtet werden. Die speziell auf die jeweiligen Körper angepassten Künstlichen Neuronalen Netze konnten nicht so kombiniert werden, dass eine geregelte Bewegung sichtbar wurde. Weiterhin ist eine Kombination zweier, unterschiedlicher großer Künstlicher Neuronaler Netze nur schwierig möglich. [Col11]

Mit diesen Beobachtungen wurde die Vermutung, dass ein Crossover zweier Agenten nicht erfolgreich sein dürfte, bestätigt. Aus diesem Grund

wurde auf die Rekombination zweier Genome verzichtet und ausschließlich die Fortpflanzung über Duplikation des Erbguts (mit anschließender Mutation) implementiert. Es wäre zwar möglich, dass aus zwei Agenten ein Nachfahre mit hoher Fitness hervorgeht. Die Wahrscheinlichkeit dafür ist aber bei zwei vollkommen unterschiedlichen Agenten aufgrund der oben dargestellten Probleme relativ gering.

An dieser Stelle soll allerdings darauf hingewiesen werden, dass eine Rekombination auch bei komplexeren Wesen durchaus vorstellbar ist. Die Natur bietet dazu zahlreiche Beispiele. Komplexeste Lebewesen wie der Mensch pflanzen sich zweigeschlechtlich fort. Der Schlüssel dazu ist das Vorhandensein eines geeigneten Genoms (der DNS), das die Rekombination erst ermöglicht. Um eine Rekombination in der Simulation durchführen zu können müsste ein geeignetes Genom entwickelt werden, mit welchem die Ähnlichkeit zweier Agenten zueinander bestimmt werden kann. Nur ähnliche Agenten, deren Eigenschaften sich nicht zu stark unterscheiden dürften dann gekreuzt werden. Damit würde sich die Wahrscheinlichkeit, erfolgreiche Nachkommen zu erhalten, vermutlich drastisch erhöhen.

### 3.3. Implementierung eigener Klassen

Die im Rahmen dieser Untersuchung durchgeführten Experimente erforderten die Entwicklung mehrerer eigener Klassen, welche im EAS-Framework verwendet werden, um die Evolution simulieren zu können. Die Klassen setzen dabei die vorhergegangenen Überlegungen und Entwürfe um. Die wichtigsten für die Experimente programmierten Klassen und deren Funktionsweise sollen hier kurz vorgestellt und erläutert werden.

**EvolvableBoxAgent3DEnvironment** Im EAS-Framework müssen alle Simulationen innerhalb einer Umgebung, eines Environments, stattfinden. Für die Simulationen, welche im Rahmen dieser Arbeit durchgeführt wurden, wurde die Klasse *EvolvableBoxAgent3DEnvironment* entwickelt. Mit dieser Klasse lässt sich eine Physik-Welt erzeugen, in welcher ähnliche Bedingungen, wie sie auf der Erde herrschen, simuliert werden können.

Die Welt besteht aus einem Raum mit einem flachen Boden, welcher einer hohen Reibung besitzt und komplett starr ist. Die Gravitation ist mit der Erdbeschleunigung vergleichbar. Da das verwendete Plugin zur visuellen Darstellung (*VideoPlugin3D*) keine Texturen unterstützt, wurde ein alternativer Weg gewählt, um den Boden mit einem Muster zu versehen, damit die Bewegung der Wesen besser erkennbar ist. Der gesamte Boden besteht aus einer Vielzahl an Quadern mit quadratischer Grundfläche, die schachbrettförmig angeordnet sind. Die Grafik-Engine stellt benachbarte Objekte in verschiedenen Farben dar, so dass die schachbrettförmig angeordneten Boden-Boxen unterschiedlich gefärbt sind. Dadurch lässt sich die Bewegung der Agenten besser wahrnehmen als auf einem strukturlosen Boden.

**EvolvableBoxAgent3D** Die für die Simulation verwendeten Agenten setzen sich – wie bereits beschrieben – aus mindestens zwei Quadern, welche miteinander durch Gelenke verbunden sind, zusammen. Der erste Quader, aus welchem ein Agent besteht, ist der Kern. Mit dieser Kernbox sind weitere Boxen, auch Anhängsel genannt, verbunden, welche vom Agenten verwendet werden können, um sich fortzubewegen.

Der eigentliche Kern des Agenten wird durch die Klasse *EvolvableBoxAgent3D* beschrieben, die Anhängsel werden durch eine weitere Klasse im Agenten gespeichert. Mit der Klasse *EvolvableBoxAgent3D* kann aber der gesamte Agent durch den Scheduler erzeugt und dem Environment hinzugefügt werden. Über die Klasse sind – direkt innerhalb der Klasse oder indirekt über Referenzen – alle Informationen und Funktionen eines Agenten abrufbar. Dazu zählen u.a. das Genom, das Künstliche Neuronale Netz, die Sensoren und Aktoren sowie alle Boxen und deren Eigenschaften.

Die Klasse stellt weiterhin Funktionen zur Beurteilung der Fitness und zum Vergleich der Fitness zweier Agenten zur Verfügung. Da die Eigenschaften des Agenten durch sein Genom vorgegeben werden, kann ein stark vereinfachter Konstruktor zur Erzeugung eines Agenten verwendet werden. Dieser Konstruktor wird im Rahmen der Simulation von einer später vorgestellten Hilfsklasse im Scheduler aufgerufen, um den Agenten mit seinen



Boxen, Sensoren, Aktoren und seinem Künstlichen Neuronalen Netz zu erzeugen.

**EvolvableBoxAgent3DAppendixInfo** In der Klasse *EvolvableBoxAgent3DAppendixInfo* werden alle Informationen über ein Anhängsel eines Agenten gespeichert. Dazu zählen die interne Identifikationsnummer einer Box, ihr Vorgänger, die Größe sowie die Position auf dem Vorgänger und die Achse, welche im Gelenk frei beweglich ist.

**EvolvableBoxAgent3DAppendix** Mit der Klasse *EvolvableBoxAgent3DAppendix* kann aus den in *EvolvableBoxAgent3DAppendixInfo* gespeicherten Informationen eine an einem Agenten oder einem anderen Quader befestigte Box erzeugt werden. Die Klasse übernimmt dabei die Aufgabe, aus den Informationen einen Quader zu erzeugen, welcher in der Physik-Engine simuliert wird.

Dieser Quader wird über ein, durch den Konstruktor initialisiertes, Gelenk mit seinem Vorgänger verbunden, so dass ein zusammenhängendes Wesen erzeugt wird. Innerhalb des Konstruktors werden dabei sämtliche Objekte erzeugt und alle Methoden aufgerufen, die jeweils nötig sind, um die Box an der richtigen Stelle mit ihrem Vorgänger zu verbinden und in der Physik-Welt zu platzieren. Dies vereinfacht wiederum die gesamte Erzeugung des zu simulierenden Wesens, da die Zahl an nötigen Methodenaufrufen bzw. Befehlen im Scheduler stark reduziert wird.

Außer der Platzierung der Box in der Physik-Welt erfolgt noch die Initialisierung der zum Gelenk gehörenden Aktoren und Sensoren. So sind die Referenzen darauf zwar implementierungstechnisch dezentral in den Anhängseln gespeichert, können aber auch aus der Klasse *EvolvableBoxAgent3D* abgerufen werden.

**EvolvableBoxAgent3DConnectable** Das Interface *EvolvableBoxAgent3DConnectable* gibt Eigenschaften und Methoden vor, die ein Objekt besitzen muss, um mit dem eigentlichen Agenten, dem Kern, verbunden werden zu

können. Das Interface wurde erstellt, um die Verwendung anderer Typen von Anhängseln zu ermöglichen.

Eine neu erzeugte Klasse, welche das Interface implementiert, könnte dazu verwendet werden, andere, möglicherweise komplexere Anhängsel zu simulieren. Es wäre ebenso denkbar, Anhängsel mit anderen geometrischen Formen zu programmieren und zur Verfügung zu stellen. Im Rahmen dieser Arbeit wurden allerdings nur einfache, quaderförmige Erweiterungen untersucht.

**EvoBoxNeuroBrain\***<sup>33</sup> Die Steuerungseinheit – das Gehirn des Agenten – wird durch die Klasse *EvoBoxNeuroBrain* modelliert. In dieser Klasse ist die Methode *decideAndReact()* von großer Bedeutung. Diese liest die Eingabewerte der angeschlossenen Sensoren aus und gibt sie an das Künstliche Neuronale Netz weiter. Nachdem das Künstliche Neuronale Netz die Berechnungen abgeschlossen hat, gibt die Methode die berechneten Ausgabewerte an die Aktoren weiter. Diese Klasse ist somit die Schnittstelle zwischen dem Körper des Agenten und seinem Künstlichen Neuronalen Netz.

**EvoBoxSparseNet\*** Das Künstliche Neuronale Netz, welches sich im Gehirn des Agenten befindet, wird durch diese Klasse dargestellt. Es wird nicht direkt vom Scheduler angesprochen sondern nur innerhalb der Klasse *EvoBoxNeuroBrain* verwendet. Weiterhin verfügt die Klasse über Mutationsoperatoren, die von der Klasse *EvolvableBoxAgent3DGenomeHandler* verwendet werden können, um das Künstliche Neuronale Netz zu mutieren. Wichtig ist, zu erkennen, dass die Klasse *EvoBoxNeuroBrain* das Künstliche Neuronale Netz (die Klasse *EvoBoxSparseNet*) beinhaltet und ansteuert und dass das Künstliche Neuronale Netz alleine den Agenten nicht steuern könnte.

---

<sup>33</sup>Die mit einem Asterisk (\*) versehenen Klassen bauen auf den von Colling, Müller und Nagel [Col11, Mü10, Nag10] geschriebenen Klassen auf.

**EvoBoxNeuron\*** Das Künstliche Neuronale Netz besteht vereinfacht ausgedrückt aus einzelnen Neuronen, die über Synapsen miteinander verbunden sind. Ein solches Neuron wird durch die Klasse *EvoBoxNeuron* repräsentiert. Die Klasse stellt Methoden zur Verfügung, um Neuronen über Synapsen miteinander zu verbinden oder voneinander zu trennen. Mutationsooperatoren, die von der Klasse *EvolvableBoxAgent3DGenomeHandler* angesprochen werden können, sind ebenfalls vorhanden.

**EvoBoxLink\*** Die Synapsen des Künstlichen Neuronalen Netzes werden durch die Klasse *EvoBoxLink* modelliert.

**EvoBoxBrainEncoding2\*** Die Klasse *EvoBoxBrainEncoding2* verfügt über eine Methode, welche genutzt werden kann, um das Gehirn des Agenten abzuspeichern. Wenn das Genom des Agenten geändert werden sollte, könnte an dieser Stelle die Art und Weise der Speicherung des Gehirns angepasst werden.

**EvolvableBoxAgent3DSensorAngle** Jedes Gelenk eines Wesens verfügt über einen eigenen Sensor, welcher den Winkel, in dem sich das Gelenk befindet, bestimmen kann. Die Funktionalität des Sensors wird über die Klasse *EvolvableBoxAgent3DSensorAngle* zur Verfügung gestellt. Dafür befindet sich in jedem Anhängsel-Objekt (*EvolvableBoxAgent3DAppendix*) die Referenz auf eine Instanz vom Typ *EvolvableBoxAgent3DSensorAngle*, die den Winkel für das Gelenk zwischen dem Anhängsel und dessen Vorgänger bestimmt.

**EvolvableBoxAgent3DSensorMovement** Der Sensor *EvolvableBoxAgent3DSensorMovement* ist im Gegensatz zum Winkel-Sensor (*EvolvableBoxAgent3DSensorAngle*) pro Wesen nur ein einziges Mal vorhanden. Er misst die aktuelle Bewegungsrichtung der Kernbox. Die anderen Boxen gehen dabei nicht in die Messung mit ein.

**EvolvableBoxAgent3DSensorOrientation** Gleiches gilt auch für die Klasse *EvolvableBoxAgent3DSensorOrientation*. Ein Wesen verfügt über einen einzigen Sensor dieser Klasse, der die Lage des Kerns im Raum misst.

**EvolvableBoxAgent3DActuatorServo** Der zurzeit einzige Aktor, welcher verfügbar ist, wird durch die Klasse *EvolvableBoxAgent3DActuatorServo* beschrieben. Wie die Winkelsensoren (*EvolvableBoxAgent3DSensorAngle*) sind auch diese Aktoren in jedem Gelenk vorhanden. Dazu wird in jedem Anhänger die Referenz auf ein Objekt vom Typ *EvolvableBoxAgent3DActuatorServo* gespeichert. Der Servo ermöglicht es, den Winkel des Gelenks zwischen einem Anhänger und dessen Vorgänger einzustellen.

**EvolvableBoxAgent3DGenome** In der Klasse *EvolvableBoxAgent3DGenome* werden sämtliche Informationen über einen Agenten gespeichert. Diese beinhalten die Größe und relative Position der einzelnen Boxen, die Verbindungen zwischen diesen, aber auch das Künstliche Neuronale Netz, welches zur Steuerung verwendet wird. Mit der *toString()*-Methode lässt sich aus dem Genom-Objekt eine textuelle Repräsentation in Form eines *Strings* erzeugen, welcher menschenlesbar ist.

**EvolvableBoxAgent3DGenomeHandler** In einem Genom sind alle verfügbaren Informationen enthalten, um aus diesem ein Wesen, welches in der Physik-Welt simuliert werden kann, zu erzeugen. Die Klasse *EvolvableBoxAgent3DGenomeHandler* stellt eine Vielzahl an Methoden rund um diese Genome zur Verfügung.

Zu Beginn jeder Simulation wird eine zufällige Population erzeugt. Dies bedeutet, dass eine gewünschte Zahl an Genomen zufällig generiert werden muss. Die Klasse enthält eine Methode, die in Abhängigkeit der übergebenen Parameter zufällige Genome erzeugt, aus welchen anschließend simulierbare Wesen geschaffen werden können.

Es ist eine Vielzahl an Befehlen und Methoden nötig, um aus den nötigen Informationen die einzelnen Boxen zu erzeugen, diese miteinander zu verbinden, das Künstliche Neuronale Netz zu erstellen, dieses mit den richtigen

Sensoren und Aktoren zu verknüpfen und schließlich das gesamte Wesen der Simulationsumgebung hinzuzufügen. Diese Aufgabe übernimmt eine weitere Methode der Klasse. Dabei werden aus dem Genom alle Eigenschaften extrahiert und sämtliche Objekte erzeugt und Methoden aufgerufen. Damit ist im Scheduler nur ein einziger Befehl nötig, um aus einem vorliegenden Genom einen Agenten der Simulationsumgebung hinzuzufügen.

Die verwendeten Genome können entweder als Objekt (beschrieben durch die Klasse *EvolvableBoxAgent3DGenome*) vorliegen oder als einfacher, menschenlesbarer *String*. Da die Genome in den Experimenten zu Auswertungszwecken als Zeichenkette gespeichert werden, ist eine weitere Methode in der Klasse vorhanden, welche das Erzeugen von Genom-Objekten aus diesen erlaubt. So können stets menschenlesbare Repräsentationen der Genome verwendet werden, ohne dass die Simulation dadurch beeinflusst würde.

Da im Rahmen der Evolution Mutationen stattfinden, wurde eine Methode, welche übergebene Genome mutiert, programmiert. Diese Methode erzeugt aus einem Genom eine mutierte Variante, wobei die Mutation entweder auf physischer Ebene (Anzahl, Position oder Größe der Boxen) oder auf neuronaler Ebene (Anzahl der Neuronen oder Gewichtung der Synapsen) abläuft. Die konkreten Mutationen werden in Abschnitt 3.2.3.1 dieser Arbeit genauer erläutert.

Zuletzt bietet *EvolvableBoxAgent3DGenomeHandler* noch Funktionen zum Speichern und Laden ganzer Populationen. Eine Population ist dabei eine Menge an *String*-Repräsentationen von Genomen.

**EvolvableBoxAgent3DSelectionMechanism** Im Rahmen der Evolution werden verschiedene Agenten ausgewählt, welche ihr Genom zu Verfügung stellen dürfen, um Nachkommen zu erzeugen. Die Klasse *EvolvableBoxAgent3DSelectionMechanism* enthält eine Methode, die aus einer Population von Agenten diejenigen auswählt, die sich fortpflanzen dürfen und deren Genome speichert.

Innerhalb dieser Klasse können weitere Selektionsmechanismen programmiert werden, so dass eine Auswahl verschiedener Mechanismen möglich ist. Somit wäre es möglich, verschiedene Mechanismen zu untersuchen und zu

vergleichen. Im Scheduler muss der Funktion nur die gesamte Population und die gewünschte Selektionsmethode übergeben werden, so dass ein hohes Maß an Flexibilität und Anpassbarkeit der Simulation bei gleichzeitiger einfacher Verwendung gewährleistet ist.

**EvolvableBoxAgent3DReproductionMechanism** Neben einer Klasse für diverse Selektionsmechanismen gibt es auch noch eine Klasse, in welcher verschiedene Reproduktionsmechanismen gespeichert werden können. Auch hier ermöglicht die Verwendung einer Klasse, welche die Auswahl des Reproduktionsmechanismus ermöglicht, eine hohe Flexibilität bei gleichzeitiger einfacher Anwendung.

Falls gewünscht, könnten hier für weitere Experimente neue Reproduktionsmechanismen programmiert werden, so dass ein Vergleich verschiedener Reproduktionsschemata möglich ist.

**EvolvableBoxAgent3DSchedulerBrain** Der Scheduler ist die dritte der drei Hauptkomponenten Environment, Agent und Scheduler einer jeden Simulation im EAS-Framework. Die Klasse *EvolvableBoxAgent3DSchedulerBrain* übernimmt dabei die Aufgabe der Steuerung des Ablaufs der in dieser Diplomarbeit durchgeführten Experimente.

In Abhängigkeit der Parameter werden durch den Scheduler zufällige Genome erzeugt, welche die Startpopulation bilden. Aus diesen Genomen werden schließlich Agenten geschaffen, welche nacheinander in einer Physik-Welt, dem Environment, simuliert werden. Nachdem alle Agenten simuliert wurden, werden über einen Selektionsmechanismus (*EvolvableBoxAgent3DSelectionMechanism*) die Eltern ausgewählt, aus welchen die nächste Generation erzeugt wird (*EvolvableBoxAgent3DReproductionMechanism*). Diese Generation wird ebenfalls simuliert und so weiter. Der genaue Ablauf der Simulation wird im nächsten Kapitel näher beleuchtet.

**EvolvableBoxAgent3DSchedulerGenomeWithBrainPresenter** Der Scheduler ermöglicht die grafische Darstellung von Agenten, welche aus einer festzulegenden Population erzeugt werden. Die Ansteuerung erfolgt dabei

genau wie in *EvolvableBoxAgent3DSchedulerBrain*, so dass theoretisch das Verhalten eines Agenten wiedergegeben werden kann. In Abschnitt 4.4 wird erklärt, warum dies nicht immer der Fall ist.

**EvolvableBoxAgent3DStatistics** Während der Durchführung einer Simulation erstellt der Scheduler mithilfe der Klasse *EvolvableBoxAgent3DStatistics* ein Protokoll des Simulationsablaufs. Zu Beginn werden alle Simulationsparameter abgespeichert, damit das Experiment nachvollzogen werden kann. Die erreichten maximalen und durchschnittlichen Fitness-Werte einer jeden Generation werden ebenso in einem Protokoll gespeichert wie das beste Genom. Mit diesen Informationen lässt sich jeder Simulationslauf auswerten und interpretieren. Sämtliche Werte werden als CSV-Dateien<sup>34</sup> abgespeichert, so dass sie anschließend mit einem Tabellenverarbeitungsprogramm geöffnet werden können.

**EvolvableBoxAgent3DTrack** Um bei Simulationen ohne direkte dreidimensionale grafische Darstellung die Bewegung der Agenten nachvollziehen zu können, wurde diese Klasse programmiert. In ihr können die einzelnen Punkte, an denen sich ein Agent zu diskreten Zeitpunkten seiner Evaluation befindet, aufgezeichnet werden.

**EvolvableBoxAgent3DLogger** Die Klasse *EvolvableBoxAgent3DLogger* erzeugt aus gespeicherten Track-Punkten eines *EvolvableBoxAgent3DTracks* eine Grafik vom Dateityp PNG<sup>35</sup>, in welcher der gesamte zurückgelegte Pfad des Agenten eingezeichnet ist. So lässt sich auch ohne das *VideoPlugin3D* die Bewegung beobachten und interpretieren.

---

<sup>34</sup>CSV steht für *Comma-Separated-Values*, ein Dateiformat, in welchem einzelne Werte durch ein Zeichen, i.d.R. ein Komma, voneinander getrennt werden.

<sup>35</sup>Bei PNG-Dateien handelt es sich um verlustlos komprimierte Rastergrafiken. PNG steht für *Portable Network Graphics*.





## 4. Experimente

Alles, was im Weltall  
existiert, ist die Frucht von  
Zufall und Notwendigkeit.

---

*Demokrit*

Im Rahmen dieser Diplomarbeit sollte die Entwicklung sich fortbewegender Wesen in einer dreidimensionalen Welt untersucht werden. Ziel der durchgeführten Experimente war, durch Evolution gleichzeitig den physischen Aufbau der Wesen und deren Künstliche Neuronale Netze so zu evolvieren, dass sich selbstständig fortbewegende Agenten entstehen. Zur Untersuchung dieser Entwicklung wurden Experimente entworfen und durchgeführt, die in den folgenden Abschnitten vorgestellt werden.

### 4.1. Entwurf der Experimente

#### 4.1.1. Probeläufe und Entwicklung der Experimente

Die verwendeten Klassen wurden über einen längeren Zeitraum entwickelt und verbessert. Währenddessen wurde eine große Anzahl an Simulationen und Probeläufen durchgeführt.

Die Probeläufe dienten dem Finden und Korrigieren von Fehlern sowie der Überprüfung verschiedener Ideen, Möglichkeiten und Simulationsvarianten. Diese testweise durchgeführten Experimente waren zum Teil fehlerhaft oder lassen sich nicht mehr nachvollziehen, da viele Änderungen an

der Simulationsumgebung vorgenommen wurden. Daher sei an dieser Stelle nur kurz ein Überblick über die durchgeführten Probeläufe gegeben.

### 4.1.1.1. Testphase und Auswahl der geeigneten Gelenke

Im Anschluss an die erfolgreiche Implementierung der 3D-Physik-Engine in das EAS-Framework wurden die vereinfachenden abstrakten Klassen programmiert, die im vorherigen Kapitel vorgestellt wurden. Mit diesen wurden zunächst diverse Physik-Tests durchgeführt, um die Tauglichkeit der Engine zu testen. Neben einfachen Versuchen, bei denen einzelne Körper in der Physik-Welt simuliert wurden, konnten auch aufwendigere Simulationen durchgeführt werden.

Dazu gehörte u.a. das Erproben der verschiedenen Gelenke und Constraints, die in JBULLET verfügbar waren. Nachdem alle Gelenkklassen testweise verwendet worden waren, blieb schließlich ein Gelenk, welches sechs Freiheitsgrade ermöglicht, übrig.<sup>1</sup> Nur dieses erfüllte die nötigen Ansprüche zufriedenstellend und war mächtig und variabel genug, um verwendet werden zu können. Während der durchgeführten Tests fielen zudem die Probleme bei zu großen Auslenkungen gegenüber der Nullstellung auf. Dies hatte die Festlegung des für das Gelenk verfügbaren Bereichs auf  $-85^\circ$  bis  $+85^\circ$  zur Folge.<sup>2</sup>

### 4.1.1.2. Entwicklung erster Agenten

Auf den im letzten Kapitel dargelegten Überlegungen beruhend wurde dann ein erster Agent mit dem oben genannten Gelenk entwickelt und erprobt. Zu Beginn verfügte dieser noch nicht über ein Künstliches Neuronales Netz, Sensoren oder Aktoren. Stattdessen wurde der Winkel des Constraints über eine Sinusfunktion eingestellt, um die korrekte Funktionsweise verifizieren zu können. Passende Mutationsoperatoren wurden ebenso implementiert wie weitere Hilfsklassen, die bereits vorgestellt wurden.

---

<sup>1</sup>`com.bulletphysics.dynamics.constraintsolver.Generic6DofConstraint`

<sup>2</sup>Vgl. für nähere Erklärungen hierzu Unterabschnitt 3.2.1.2.

In einem ersten größeren Versuchslauf wurden anschließend Agenten ohne Künstliches Neuronales Netz evolviert. Die Agenten veränderten lediglich kontinuierlich und synchron den Winkel aller Gelenke, so dass gleichmäßig der gesamte Weg von der Nullstellung des Gelenks bis zu einem Maximalausschlag, zum anderen Maximalausschlag und wieder zurück erfolgte. Diese Winkeländerung reichte aus, um die Agenten zu bewegen.

Die durchgeführten Mutationen veränderten ausschließlich den physischen Aufbau der Agenten. Nach mehreren Generationen zeichnete sich eine deutliche Fitnesssteigerung ab. Dieser Versuch zeigte, dass alleine durch die Veränderung der physischen Eigenschaften und mit einem äußerst simplen Steuerungsmechanismus eine Verbesserung der Agenten möglich war.

### 4.1.1.3. Einbeziehung Künstlicher Neuronaler Netze

An den Versuchslauf mit einem primitiven Gehirn anknüpfend sollte schließlich ein Künstliches Neuronales Netz implementiert werden. Da im EAS-Framework bereits Klassen mit einem Künstlichen Neuronalen Netz vorhanden waren<sup>3</sup>, mussten diese lediglich angepasst werden. Es wurden weitere Sensoren entwickelt, die Eingabewerte liefern können. Die Ausgabewerte wurden so normiert, dass sie mit den Aktoren kompatibel waren. Anschließend konnten erste Versuche mit „intelligenten“ Agenten durchgeführt werden. Nachdem das Zusammenspiel von Gehirn und Körper der Agenten ausreichend getestet worden war, wurden erste Probeversuche mit gleichzeitiger Evolution von physischem Aufbau und Künstlichem Neuronalen Netz durchgeführt.

### 4.1.1.4. Lösung von Problemen bei der Simulation

Das Steuern eines Agenten in einer dreidimensionalen Physik-Welt mit einem Künstlichen Neuronalen Netz ist ein komplexes Problem, welches viel Potential für Fehler bietet. Während der Experimente wurden immer wieder Verbesserungen und Optimierungen durchgeführt. So konnte beispiels-

---

<sup>3</sup>BENEDIKT MÜLLER, CHRISTIAN NAGEL UND DOMINIK COLLING leisteten hierzu die Vorarbeiten. [Mü10, Nag10, Col11]

weise die ursprünglich benötigte Simulationszeit um mehr als die Hälfte reduziert werden. Es tauchten allerdings auch unerwartete Fehler auf. Beispielhaft sei hier ein solches Problem geschildert.

Zu Beginn eines jeden Experiments wurden die Agenten auf dem Boden platziert und ihr Gehirn aktiviert. Dies funktionierte häufig ohne Probleme. Manche Agenten sprangen jedoch sofort außerordentlich weit vom Startpunkt weg und erhielten eine sehr hohe Fitness, nur um sich anschließend nicht mehr zu bewegen.

Dafür gibt es folgenden Grund. Da sich die einzelnen Boxen eines Wesens zu Beginn nicht zwangsweise in der richtigen Position befinden, kann es sein, dass nicht alle durch die Gelenke vorgegebenen Constraints erfüllt sind. Wenn das Wesen nicht „in der Luft“, sondern auf dem Boden der Physik-Welt platziert wird, können nicht alle Constraints ohne Probleme von der Physik-Engine durchgesetzt werden. Die Constraints werden von der Physik-Engine zwar durch angelegte Kräfte und Drehmomente erfüllt, dies hat jedoch zur Folge, dass das Wesen vom Boden abgestoßen wird und dadurch möglicherweise vom Startpunkt weg fliegt.

Wenn das Einfügen in einer gewissen Höhe und nicht auf dem Boden erfolgt, tritt dieses Problem nicht mehr auf. Das Wesen „entspannt“ sich während des freien Falls und alle Boxen befinden sich schließlich in einer Position, in welcher die Gelenk-Constraints erfüllt sind. Erst dann kann die Fitnessmessung starten.

Für das Einfügen, den Fall und das Entspannen haben sich zehn Sekunden als guter Richtwert erwiesen. Die Zeit ist auch für komplexere Wesen ausreichend, so dass diese entspannt und ohne weitere Bewegungen auf dem Boden ruhen. In der Regel wären nur wenige Sekunden nötig, die Erweiterung auf zehn Sekunden lässt aber ausreichend Spielraum. Der zusätzliche Rechenaufwand für diese Maßnahme hält sich des Weiteren in Grenzen, da eine Simulation von sich nicht (mehr) bewegenden Körpern in JBULLET sehr schnell erfolgt.

Probekalber durchgeführte Experimente ohne diese Methode haben stets zu unerwünschten Ergebnissen in Form von „defekten“ Agenten geführt. Es bildeten sich häufig Wesen aus, welche zu Beginn bereits starke Ver-

letzungen der Constraints hatten. Um die Constraints möglichst schnell durchsetzen zu können, legt die Physik-Engine sehr hohe Kräfte und Drehmomente an. Dies hat wiederum wie beschrieben zur Folge, dass der Agent sehr starke Kräfte erfährt. Aus diesem Grund wurden manche Agenten alleine durch die Physik-Engine weit geschleudert und erhielten eine hohe Fitness, ohne sich überhaupt selbst bewegt zu haben. Da dies nicht dem gewünschten Evolutionsziel entsprach, wurde die soeben erläuterte Maßnahme eingeführt und der zusätzliche Rechenaufwand in Kauf genommen.

Neben der Beseitigung weitere Probleme und Unsauberkeiten wurden die Versuche weiter verfeinert und abgeändert bis schließlich der finale Versuchsablauf entworfen war.

### 4.1.2. Grundsätzlicher Ablauf eines Experiments

Aufbauend auf den Erfahrungen und Überlegungen der Probeläufe und Vorversuche wurde schließlich ein Versuchsablauf entwickelt, der die Untersuchung der gleichzeitigen Evolution von Körper und Gehirn der Wesen ermöglicht. Dieser Versuchsablauf soll im Folgenden vorgestellt und erläutert werden. Die für diese Arbeit durchgeführten Experimente wurden alle durch denselben Scheduler<sup>4</sup> gesteuert, der über verschiedene Parametereinstellungen unterschiedliche Abläufe ermöglicht. Der Scheduler implementiert den entworfenen Versuchsablauf und führt damit die Experimente durch.

Eine Simulation läuft – natürlich abhängig von den gewählten Parametern – prinzipiell immer gleich ab. Zu Beginn wird über die Parameterliste der sogenannte *random seed* ausgelesen, mit welchem ein Zufallszahlengenerator erzeugt wird. Bei dem *random seed* handelt es sich um eine Zahl vom Datentyp *long*, die als Startparameter für den Pseudozufallszahlengenerator dient. Der Generator erzeugt keine echten Zufallszahlen sondern lediglich Pseudozufallszahlen in Abhängigkeit von dem übergebenen Startparameter.

Es handelt sich deshalb um *Pseudozufallszahlen*, da ein bestimmter Start-

---

<sup>4</sup>EvolvableBoxAgent3DSchedulerBrain

parameter immer zu denselben Ausgaben führt. Dies ist für die Durchführung der Experimente ausdrücklich erwünscht. So lassen sich alle durchgeführten Simulationsläufe exakt nachvollziehen, da alle „zufälligen“ Zahlen und Werte auf dieselbe Art und Weise generiert werden. Werden zwei Instanzen eines Experiments mit jeweils denselben Parametern, insbesondere mit demselben *random seed*, gestartet, so sind der Ablauf und das Ergebnis der Experimente exakt gleich. [Col11]

Im Anschluss wird mit dem initialisierten Pseudozufallszahlengenerator die Startpopulation erzeugt. Dazu wird die vom Benutzer gewünschte Populationsgröße ausgelesen und eine entsprechende Anzahl an zufälligen Genomen erzeugt. Dies geschieht über eine Hilfsklasse<sup>5</sup>. Bei der Erzeugung der Genome gehen ebenfalls wieder vom Benutzer festzulegende Parameter mit ein. So muss der Benutzer beispielsweise festlegen, über wie viele Boxen ein Genom zu Beginn maximal verfügen darf und welche Sensoren verfügbar sein sollen. Neben einer zufälligen Körperform wird auch ein zufälliges, nicht-leeres Künstliches Neuronales Netz erzeugt, so dass bereits zu Beginn eine Steuerung der Gelenke möglich sein kann.

Nach dieser Initialisierung startet die eigentliche Simulation der aus den Genomen erzeugten Agenten. Alle in der Population vorhandenen Genome werden nacheinander ausgelesen. Aus einem ausgelesenen Genom wird ein Agent erzeugt, welcher in der Physik-Welt simuliert wird. Dazu wird der Agent gemäß der Informationen im Genom in der dreidimensionalen Welt zusammengesetzt und aufgebaut. Dies geschieht in großer Höhe über dem Boden der Physik-Engine, um Probleme mit nicht erfüllten Constraints zu vermeiden. Die Simulation läuft für zehn Sekunden, ohne dass das Gehirn des Agenten aktiviert wird. Der Agent wird durch die Gravitation in Richtung Boden gezogen und die Boxen bewegen sich währenddessen an die von den Gelenken vorgegebenen Positionen.

Sobald die Zeitspanne von zehn Sekunden simuliert wurde, sollte sich der Agent in einem Ruhezustand auf dem Boden befinden und sich nicht mehr bewegen. Die Position, an welcher sich der Agent nun befindet, wird

---

<sup>5</sup>EvolvableBoxAgent3DGenomeHandler

als Startposition für die Fitnessmessung gespeichert. Dieser Startpunkt wird bestimmt, indem der Durchschnitt aller Schwerpunkte der Boxen eines Agenten gebildet wird. Erst danach wird das Künstliche Neuronale Netz aktiviert und die Bewertung des Agenten beginnt. Das Künstliche Neuronale Netz erhält über das Gehirn des Agenten von den Sensoren die Eingabewerte und berechnet die Ausgabewerte für die Aktoren. Die Berechnungen werden mit einer bestimmten, über einen Parameter festgelegten, Frequenz ausgeführt. Ein funktionierendes Künstliches Neuronales Netz führt dann dazu, dass in Abhängigkeit der Sensorwerte die richtigen Aktorwerte berechnet werden, so dass sich der Agent vom Startpunkt weg bewegt.

Am Ende der vom Benutzer festgelegten Zeit für einen Bewertungslauf wird erneut der Durchschnitt der Schwerpunkte aller Boxen eines Agenten bestimmt. Die Fitness entspricht der Entfernung der erreichten Endposition vom zuvor gespeicherten Startpunkt. Dabei gehen lediglich die x- und die z-Komponente der Positionen in die Berechnungen mit ein. Die y-Komponente<sup>6</sup> wird nicht mit einbezogen. Es wird ausschließlich die in der Ebene zurückgelegte direkte Verbindungsstrecke zwischen Start- und Endpunkt bewertet. Je weiter diese auseinanderliegen, desto höher ist die Fitness eines Agenten.

Am Ende eines jeden Bewertungslaufs wird der Agent aus der Physik-Welt entfernt und diese wird zurückgesetzt. Dies ist nötig, um eine Wiederholbarkeit der Bewertung zu gewährleisten. Nur so ist garantiert, dass ein und dasselbe Genom stets dieselbe Fitness erlangt.<sup>7</sup>

Auf die beschriebene Art und Weise wird aus jedem Genom der Population jeweils ein Agent erzeugt und bewertet. Am Ende einer Generation wird die bessere Hälfte der Population ausgewählt. Die Hälfte mit den niedrigeren Fitnesswerten wird verworfen. Die ausgewählten Agenten dürfen auch in der nächsten Population evaluiert werden, d.h. deren Genome werden ausgelesen und in die Population der nächsten Generation kopiert.

---

<sup>6</sup>JBULLET verfügt über ein rechtshändiges Koordinatensystem, bei welchem die y-Achse die senkrechte Achse darstellt.

<sup>7</sup>Siehe hierzu auch Abschnitt 4.4.

Außerdem wird von jedem Genom, welches ausgewählt wurde, ein Nachfolger erzeugt. Dazu wird eine Mutation am Genom vorgenommen, die dieses leicht verändert. Die Mutation kann entweder den Körper betreffen oder das Künstliche Neuronale Netz und führt im Optimalfall zu einer Verbesserung der Fitness des Agenten.

Die neu erstellte Population wird ebenfalls bewertet. Daraufhin dürfen sich die Agenten aus der besseren Hälfte der Population wieder selbst kopieren und jeweils einen mutierten Nachkommen erzeugen usw. Nach einer vom Benutzer festgelegten Zahl an Generationen entsteht so im Idealfall eine Endpopulation aus Individuen, die über eine deutlich höhere Fitness verfügen als die Individuen der Startpopulation.

Während der Durchführung der Bewertungsläufe werden Informationen wie Fitness, die Genome der besten Individuen und zurückgelegte Strecken aufgezeichnet und gespeichert. Dies ermöglicht eine Auswertung und Interpretation der Simulationsläufe. Da zudem auch noch die Genome der Endpopulation gespeichert werden können, ließen sich diese als Startpopulation für weitere Experimente verwenden.

### 4.1.3. Erklärung der Parameter

Das EAS-Framework ermöglicht die Verwendung von Parametern, um Einstellungen und Werte nicht im Quellcode festlegen zu müssen. Stattdessen ist es möglich, die Parameter vor dem Start der Simulation einzustellen und auch mehrere Experimente gleichzeitig mit unterschiedlichen Parametern laufen zu lassen. Dies vereinfacht die Verwendung von JOSCHKA, um die Simulationen über mehrere Rechner verteilt durchführen zu können.

Die Parameter lassen sich in verschiedene Gruppen unterteilen. Einige dieser Einstellungen müssen bei allen Simulationen gesetzt werden, um überhaupt die Simulationsumgebung einzurichten und beispielsweise festzulegen, welches Experiment durchgeführt werden soll. Diese allgemeinen Parameter sollen zuerst vorgestellt werden.

**masterScheduler** Mit diesem Parameter wird der auszuführende Scheduler und damit das Experiment festgelegt. Durch die Auswahl werden



möglicherweise weitere Parameter erforderlich, die für das gewählte Experiment eingegeben werden müssen.

**directory** Gibt das Verzeichnis an, in welchem die Simulation ablaufen soll. Da JOSCHKA ausschließlich flache Hierarchien erlaubt, muss dieser Parameter auf „.“ gesetzt werden. Der einzelne Punkt bedeutet hierbei, dass alle Dateien im Wurzelverzeichnis liegen. Bei Experimenten ohne JOSCHKA sind beliebige Verzeichnisse und damit auch vom Punkt abweichende Parameter erlaubt.

**actionOnUncaughtException** Beschreibt die auszuführende Aktion, wenn eine Exception auftritt und diese nicht abgefangen wird. Der Wert „*RemoveCausingPluginAndRecover*“ erlaubt das Fortführen des Experiments.

**joschkadirectory** Bestimmt das Verzeichnis, in welchem die JOSCHKA-Dateien auf dem Netzlaufwerk des *joschkausers* gesucht werden.

**joschkajar** Bezeichnet den Dateinamen des zu erzeugenden JOSCHKA-Pakets.

**joschkauser** Ist der das Experiment ausführende JOSCHKA-Benutzer.

**simulationlength** Definiert die Anzahl an zu simulierenden Ticks. Die Simulation beginnt mit dem Tick 0. Bei den für diese Arbeit durchgeführten Experimenten werden die einzelnen Ticks als Generationen interpretiert, d.h. ein Tick entspricht einer Generation.

**plugin** Dient der Aufzählung aller für die Simulation nötigen Plugins.

**seed** Fungiert als Startwert für den Pseudozufallszahlengenerator.

Neben den oben genannten, allgemeinen Parametern gibt es abhängig vom durchgeführten Experiment weitere festzulegende Einstellungen. Bei den für die vorliegende Diplomarbeit durchgeführten Experimenten sind folgende weitere Parameter zu bestimmen.

#### 4. EXPERIMENTE

---

**fileName** Gibt den Dateinamenpräfix aller Dateien an, die im Rahmen der Experimente gespeichert werden.

**folderName** Gibt den Ordernamen an, in welchem die Simulationsergebnisse gespeichert werden.

**sameParamsId** Dient dazu, mehrere Experimente mit gleichen Parametern durchführen und so Mittelwerte bilden zu können.

**durationOfOneSimulationRound** Gibt an, wie lange die Evaluation eines Agenten in Sekunden dauern soll.

**populationSize** Gibt die Anzahl an Agenten in der Population an.

**frequencyOfThoughts** Bestimmt die Anzahl an Denkprozessen pro Sekunde, die einem Agenten gestattet werden.

**sparseNetRecurrent** Gibt an, ob das verwendete Künstliche Neuronale Netz rekurrent ist.

**useOrientationSensor** Gestattet die Nutzung des Orientierungssensors.

**useMotionSensor** Gestattet die Nutzung des Bewegungssensors.

**maxNumberOfBoxesPerAgent** Definiert die maximale Anzahl an Boxen, mit denen ein Agent bei seiner zufälligen Erzeugung ausgestattet werden darf.

**portionSparseNet** Bestimmt mit *portionBody* das Verhältnis von Mutationen des Künstlichen Neuronalen Netzes zu Mutationen des Körpers.

**portionBody** Bestimmt mit *portionSparseNet* das Verhältnis von Mutationen des Künstlichen Neuronalen Netzes zu Mutationen des Körpers.

**dropInNeuronProbability** Gibt die Wahrscheinlichkeit an, ein inneres Neuron hinzuzufügen.

**deleteNeuronProbability** Gibt die Wahrscheinlichkeit an, ein inneres Neuron zu löschen.

**createLinkProbability** Gibt die Wahrscheinlichkeit an, eine neue Synapse zu erzeugen.

**destroyLinkProbability** Gibt die Wahrscheinlichkeit an, eine Synapse zu löschen.

**changeWeightProbability** Gibt die Wahrscheinlichkeit an, mit der die Methode zur Änderung der Synapsengewichte aufgerufen wird. Die Änderung der Synapsen erfolgt dann nochmals mit einer anderen Wahrscheinlichkeit.

**changeWeightOfThisLinkProbability** Gibt die Wahrscheinlichkeit an, das Verbindungsgewicht einer konkreten Synapse zu ändern. Mit dieser Wahrscheinlichkeit wird eine bestimmte Synapse verändert, sobald die Methode zur Änderung der Synapsengewichte aufgerufen wurde.

Die Vielzahl an Parametern gestattet es dem Benutzer, verschiedenste Experimente und Simulationen mit einem einzigen Scheduler durchzuführen. Es ist möglich, einzelne Werte zu variieren, um die daraus resultierenden Auswirkungen beobachten zu können.

Das EAS-Framework bietet eine Möglichkeit an, einem Parameter mehrere Werte zuzuweisen. Dazu müssen die gewünschten Werte zeilenweise getrennt in das Eingabefeld des Parameters eingetragen werden.

Ein doppelter Rechtsklick in ein Parameterfeld hat ein automatisches Vervielfachen des Parameters sowie ein Anfügen eines Suffix an diesen zur Folge. So lassen sich beispielsweise für mehrere Simulationsläufe mit verschiedenen Parametern getrennte Ordner erstellen (Ordner00, Ordner01, Ordner02, ...). Ein dreifacher Rechtsklick in das Parameterfeld *seed* führt zum Auffüllen mit zufällig erzeugten, verschiedenen *random seeds*.

Wenn für einen Parameter mehr als ein Wert festgelegt wurde, dann ist das EAS-Framework in der Lage, eine JOSCHKA-Datei zu erzeugen, die sämtliche Kombinationsmöglichkeiten aller Parameter enthält. So lassen sich mit einem einzigen Aufruf eine ganze Reihe von verschiedenen Experimenten anstoßen.

### 4.2. Durchführung und Auswertung

Die gesamte dreidimensionale Simulationsumgebung wurde von Grund auf neu entworfen und entwickelt. Die während dieser Entwicklung durchgeführten Experimente waren zum Teil fehlerhaft oder lassen sich wegen der Veränderungen an den verwendeten Klassen mittlerweile nicht mehr nachvollziehen. Außerdem bauten verschiedene Experimente häufig auf einer unterschiedlichen Grundlage auf. Aus diesem Grund wurden zum Abschluss der Arbeit eine große, konsistente Versuchsreihe gestartet, die eine nachvollziehbare Untersuchung der Evolution ermöglicht und für alle Experimente dieselbe Basis bietet.

#### 4.2.1. Abschließende Versuchsreihen

In den abschließenden Versuchsreihen wurde untersucht, ob die erstellten Agenten so evolviert werden können, dass sie sich in einer vorgegebenen Zeit möglichst weit von ihrem Startpunkt weg bewegen. Die Evolution umfasste sowohl den physischen Aufbau der Agenten als auch ihre Künstlichen Neuronalen Netze.

Um die Komplexität und die benötigte Rechenzeit möglichst gering zu halten, wurden lediglich vier verschiedene Parameter variiert. Da zu jeder der vierundzwanzig Parameterkonstellation zehn parallele Versuche laufen sollten, ergab sich eine Gesamtzahl von zweihundertvierzig Versuchen in einer Versuchsreihe. Die kumulierte Rechenzeit betrug in etwa eintausend Stunden. Durch die Verwendung von JOSCHKA war es aber möglich, alle Berechnungen einer Versuchsreihe innerhalb weniger Tage durchzuführen.

##### 4.2.1.1. Verwendete Parameter

Für die in den nächsten Unterabschnitten vorgestellten Versuchsreihen wurden jeweils die gleichen Parametersätze verwendet. Diese lassen sich in der nachstehenden Tabelle 4.1 finden. Warum sich die Versuchsreihen trotz gleicher Parameterliste unterscheiden wird in den Abschnitten der jeweiligen Versuchsreihen erläutert.

Mit einem Asterisk (\*) markierte Parameter enthalten mehrere Einträge. Diese Parameter wurden zu Versuchszwecken variiert. Eine besondere Rolle spielt weiterhin der Parameter *sameParamsId*. Er wurde eingeführt, um Versuche mit demselben Parametersatz aber unterschiedlichem *random seed* mehrmals durchführen zu können. Damit lassen sich mehrere Simulationen mit gleichen Parametern aber unterschiedlichen Startwerten für den Zufallszahlengenerator erstellen, so dass Durchschnittswerte gebildet werden können.

Tabelle 4.1.: Parameter, die in den durchgeführten Experimenten verwendet wurden

Parametername	Wert	Anmerkungen
masterScheduler	defaultmaster-fr.EvolvableBoxAgentsBrain	Standardscheduler für die hier durchgeführten Experimente
directory	.	Kann bei Experimenten ohne JOSCHKA auf anderen Wert gesetzt werden
actionOnUncaughtException	RemoveCausingPluginAndRecover	Ermöglicht Fortsetzung selbst bei Fehlern
joschkadirectory	<i>user</i> ExperimentName	
joschkajar	<i>user</i> Experiment.jar	
joschkauser	<i>user</i>	Ersetzen durch Benutzernamen in JOSCHKA
simulationlength	249	Beginn bei 0, daher 250 Generationen

#### 4. EXPERIMENTE

---

plugin	eboxgenomehandler	Plugin, um Genome einfacher verwenden zu können
seed	5685973218049341691, ...	Dreifacher Rechtsklick in Parameterfeld füllt nötige zufällige seeds auf
fileName	EvoBoxFile	
folderName	Experiment <i>xxx</i>	Doppelter Rechtsklick in Parameterfeld erzeugt automatisch alle Suffixe <i>xxx</i>
sameParamsId	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	Dient der mehrfachen Durchführung von Simulationen mit denselben Parametern
durationOfOne- SimulationRound	30.0	
populationSize*	50, 100, 150	Wird variiert, um Auswirkungen zu untersuchen
frequencyOf- Thoughts	5	

sparseNetRecurrent*	true, false	Wird variiert, um Auswirkungen zu untersuchen
useOrientationSensor*	true, false	Wird variiert, um Auswirkungen zu untersuchen
useMotionSensor*	true, false	Wird variiert, um Auswirkungen zu untersuchen
maxNumberOfBoxesPerAgent	5	
portionSparseNet	0.6	
portionBody	0.4	
dropInNeuronProbability	0.05	
deleteNeuronProbability	0.05	
createLinkProbability	0.4	
destroyLinkProbability	0.2	
changeWeightProbability	0.3	
changeWeightOfThisLinkProbability	0.7	

#### 4.2.1.2. Anmerkungen zu den gewählten Parametern

Die Populationsgröße wurde so gewählt, dass eine ausreichende Zahl an zufälligen Genomen zur Verfügung steht, um zumindest zu Beginn eines Simulationslaufs ein gewisses Maß an Diversität zu ermöglichen. Gleich-

zeitig wurden aber keine zu großen Populationen zugelassen, da dies die Rechenzeit dramatisch erhöht.

Gleiches gilt auch für die Simulationszeit. Diese wurde mit dreißig Sekunden so gewählt, dass den Agenten ausreichend Zeit zur Verfügung steht, die evolvierte Bewegungsstrategie durchzuführen und sich so möglichst weit von ihrem Startpunkt zu entfernen. Längere Zeiten hätten eine proportional längere Rechenzeit zufolge, so dass die Evaluationszeit auf die genannten dreißig Sekunden beschränkt wurde.

Neben unterschiedlichen Populationsgrößen wurde weiterhin untersucht, ob es einen Unterschied macht, ob das zugrundeliegende Künstliche Neuronale Netz ein Feed-Forward-Netz oder ob es rekurrent ist.<sup>8</sup> Zuletzt wurde noch getestet, ob das Zuschalten verschiedener Sensoren eine Verbesserung der Fitness bewirken kann oder sich möglicherweise sogar negativ auswirkt. Insgesamt wurden die Parameter *populationSize* (50, 100, 150), *sparseNet-Recurrent*, *useOrientationSensor* und *useMotionSensor* (jeweils `true` oder `false`) variiert.

### 4.2.1.3. Überblick über die Experimente

Durch die Variation der Parametern ergeben sich vierundzwanzig verschiedene Kombinationen. Somit besteht eine Versuchsreihe aus vierundzwanzig verschiedenen Experimenten, die jeweils zehn mal durchgeführt werden, um anschließend den Mittelwert der erreichten Fitness bilden zu können. Eine Übersicht über die Experimente liefert die untenstehende Tabelle.

---

<sup>8</sup>Auf den Unterschied zwischen diesen zwei Arten von Netzen wird in Abschnitt 2.3.3.2 eingegangen.



Tabelle 4.2.: Überblick über die durchgeführten Experimente

Name des Experi-ments	Parameter			
	popula- tionSize	sparse- NetRe- current	useOrien- tationSen- sor	useMo- tionSen- sor
A	50	false	false	false
B	50	true	false	false
C	100	false	false	false
D	100	true	false	false
E	150	false	false	false
F	150	true	false	false
G	50	false	true	false
H	50	true	true	false
I	100	false	true	false
J	100	true	true	false
K	150	false	true	false
L	150	true	true	false
M	50	false	false	true
N	50	true	false	true
O	100	false	false	true
P	100	true	false	true
Q	150	false	false	true
R	150	true	false	true
S	50	false	true	true
T	50	true	true	true
U	100	false	true	true
V	100	true	true	true
W	150	false	true	true
X	150	true	true	true

#### 4.2.2. Erste Versuchsreihe mit normaler Fitnessfunktion

Die erste durchgeführte Versuchsreihe erfolgte mit den in der obigen Tabelle 4.2 aufgeführten Parametern und einer normalen Fitnessfunktion.

Die Fitness entsprach dabei der Entfernung des Endpunktes vom Startpunkt in Metern. Sie gibt somit an, wie weit sich der Agent von seinem Startpunkt innerhalb der dreissigsekündigen Evaluationszeit entfernt hat. Die Start- und Endpunkte werden dabei wie zuvor beschrieben jeweils über die Durchschnittsbildung aller Boxenschwerpunkte bestimmt.

##### 4.2.2.1. Ergebnisse der ersten Versuchsreihe mit normaler Fitnessfunktion

In der nachstehenden Tabelle 4.3 sind die Durchschnitte der maximal erreichten Fitnesswerte einer Simulationsreihe zu finden. Die Durchschnittswerte wurden dabei über die jeweiligen Maximalwerte der zehn Simulationsläufe eines jeden Experiments gebildet. Die zurückgelegte Entfernung wurde auf ganze Zahlen gerundet, womit der Fitnesswert die Entfernung vom Startpunkt in Metern angibt.

Tabelle 4.3.: Erste Versuchsreihe: Maximal erreichte Fitness mit normaler Fitnessfunktion

<b>Name des Experiments</b>	<b>Maximal erreichte Fitness (Durchschnitt über je zehn Simulationsläufe)</b>
A1	3434
B1	6927
C1	6845

D1	872
E1	11504
F1	17825
G1	15416
H1	151
I1	10578
J1	5423
K1	24121
L1	40512
M1	118
N1	1383
O1	8787
P1	18823
Q1	6372
R1	2524
S1	3999
T1	2428
U1	4831
V1	4875
W1	3452
X1	14574
Durchschnitt	8991
Varianz	86664247

#### 4.2.2.2. Auswertung der Ergebnisse aus der ersten Versuchsreihe

Wie nur unschwer zu erkennen ist, wurden zum Teil enorme Fitnesswerte erreicht. Eine durchschnittliche Fitness von 11504 bedeutet beispielsweise, dass die besten Agenten durchschnittlich 11504 Meter in 30 Sekunden zurückgelegt haben müssen.<sup>9</sup> Dies entspricht wiederum etwa 383 Metern in

---

<sup>9</sup>Siehe bspw. Experiment *E*.

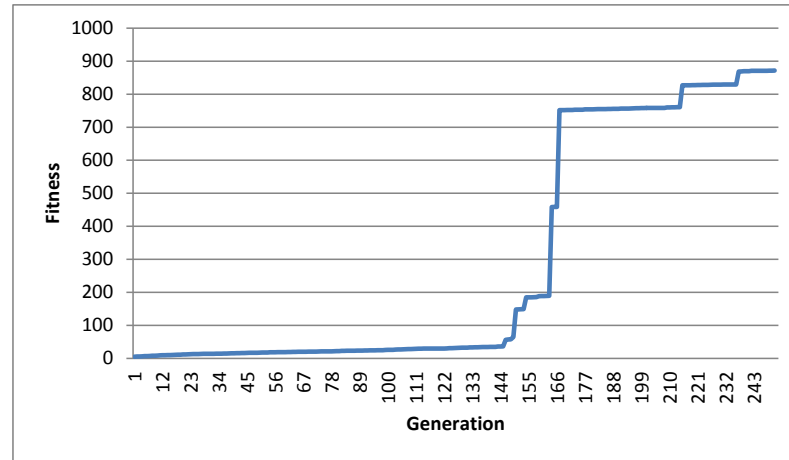


Abbildung 4.1.: Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment *D1*

einer Sekunde. Damit überschritten manche Agenten die Schallgeschwindigkeit<sup>10</sup> in der Simulation.

Zuerst wurde ein Fehler in der Fitnessfunktion vermutet, diese Möglichkeit konnte jedoch schnell verworfen werden, da zu Beginn jeder Simulation korrekte Fitnesswerte geliefert werden und auch die Pfadaufzeichnungen zu den erreichten Entfernungen passen. Unbestritten ist dennoch, dass ein Fehler oder zumindest eine Unstimmigkeit vorliegen muss, die solche hohen Fitnesswerte rechtfertigt. Zur Erklärung des Phänomens soll beispielhaft das Experiment *D* herangezogen werden.

Abbildung 4.1 repräsentiert den typischen Verlauf eines Experiments der ersten Versuchsreihe. In den ersten Generationen ist – wie erwartet – ein

<sup>10</sup>Die Schallgeschwindigkeit beträgt 343m/s in der Luft.

leichter, kontinuierlicher Anstieg der Fitness zu beobachten. Die Entwicklung der Fitnesswerte scheint normal zu verlaufen und erreicht zunächst keine unrealistischen Bereiche.

Die besten Agenten der Generation 99 legten so beispielsweise durchschnittlich 26 Meter zurück. Im schwächsten Lauf erreichte ein Agent lediglich 19,0 Meter, der beste Agent legte 32,9 Meter zurück. Diese Entfernungen entsprechen bei einer Simulationsdauer von 30 Sekunden etwa langsamer Schrittggeschwindigkeit. Somit erfüllte das Experiment *D* zumindest in den ersten Generationen die Erwartungen.

Nach etwa 150 Generationen erfolgten jedoch von Generation zu Generation plötzlich mehrere Sprünge um zum Teil deutlich über 100 Fitnesspunkte. Das bedeutet, dass die erreichte Entfernung in einigen der Simulationsläufe enorm zugenommen haben muss. Dieses Phänomen tritt jedoch nicht in allen zehn Simulationsläufen des Experiments auf, sondern lässt sich nur in einigen der Versuche beobachten. Die jeweils in der letzten Generation erreichten Fitnesswerte sind in Tabelle 4.4 zu finden.

Zwei der Simulationsläufe erscheinen besonders auffällig. So weisen sowohl Lauf 1 als auch Lauf 3 deutlich höhere Fitnesswerte als die anderen Läufe auf. Anscheinend sind diese beiden Läufe fehlerbehaftet. Durch diese beiden Läufe wird eine hohe Durchschnittsfitness von 872 mit einer sehr hohen Varianz erreicht. Wenn diese beiden Läufe ignoriert werden, liegt der Durchschnitt bei lediglich 59 und die Varianz beträgt 505.

Ein Vergleich der zurückgelegten Pfade gibt weiteren Aufschluss über das Verhalten der Agenten. In den Grafiken zeigt das grüne Quadrat den Startpunkt und das rote Quadrat den Endpunkt eines Agenten innerhalb seiner Evaluation an. Während der gesamten Evaluationsdauer wurde die Position des Agenten aufgezeichnet, wodurch die in den folgenden Abbildungen gezeigten Pfade entstehen.

Um die Unterschiede zwischen einem normalen Agenten und einem Agenten mit irregulärem Verhalten verstehen zu können, sollen die Pfade zweier verschiedener Agenten verglichen werden. Der zurückgelegte Weg des besten Agenten aus Lauf 6 des Experiments *D* ist in Abbildung 4.2 dargestellt.

Tabelle 4.4.: Maximal erreichte Fitnesswerte der einzelnen Simulationsläufe des Experiments *D1*

<b>Lauf</b>	<b>Maximal erreichte Fitness</b>
0	43
1	1089
2	49
3	7155
4	43
5	106
6	80
7	45
8	55
9	51
Durchschnitt	872
Durchschnitt ohne Läufe 1 und 3	59
Varianz	4979385
Varianz ohne Läufe 1 und 3	505

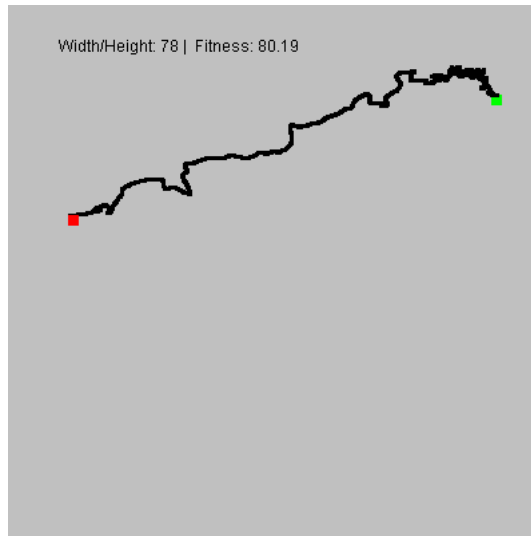


Abbildung 4.2.: Zurückgelegter Pfad des besten Agenten aus Lauf 6 des Experiments *D1*. Der Startpunkt ist grün markiert, der Endpunkt rot.

Dieser Agent hat eine normale Fitness<sup>11</sup> und zeigt ein nicht ungewöhnliches Fortbewegungsmuster. Abbildung 4.3 zeigt den Pfad des besten Agenten aus Lauf 3, einem Lauf mit stark überhöhter Fitness und irregulärem Fortbewegungsmuster.

Bereits auf den ersten Blick ist die um Größenordnungen höhere Fitness des Agenten aus Lauf 3 zu erkennen. Der vom Agenten aus Lauf 6 aufgezeichnete Pfad ist nicht geradlinig sondern weist Knicke und Kurven auf. Der Agent mit der unnatürlich hohen Fitness hingegen hat einen exakt geradlinigen Weg zurückgelegt und dabei eine sehr große Strecke überwunden.

Wie in Abbildung 4.4 zu erkennen ist, war die Fitness im betrachteten Lauf über längere Zeit im normalen Bereich. Erst nach deutlich mehr als 100 Generationen fanden in kurzer Zeit Mutationen statt, die die Fitness

---

<sup>11</sup>In keinem Experiment wurden Agenten mit nicht irregulärem Verhalten gefunden, die eine Fitness von mehr als 200 aufwiesen. Daher ist dies als grobe Richtlinie zu verstehen.

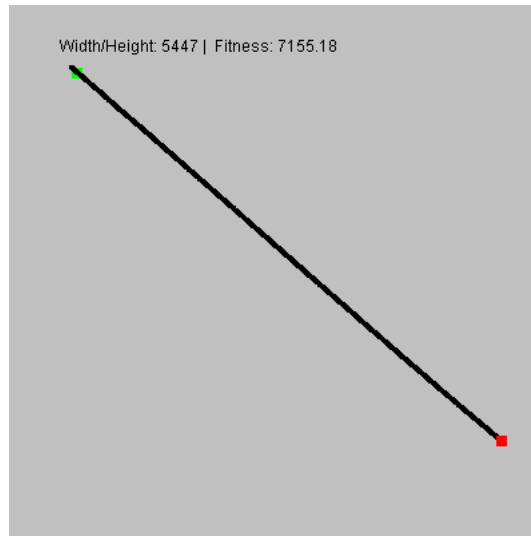


Abbildung 4.3.: Zurückgelegter Pfad des besten Agenten aus Lauf 3 des Experiments *D1*. Der Startpunkt ist grün markiert, der Endpunkt rot.

des besten Agenten aus diesem Lauf drastisch steigerten. Bemerkenswert ist hier, dass diese Mutationen in einem relativ kurzen Abstand von etwa 30 Generationen stattfinden, wobei danach keine weitere Verbesserung mehr zu beobachten ist. Dass bei zehn Simulationsläufen pro Experiment statistische Ausreißer wie der betrachtete Simulationslauf 3 für eine enorme Varianz sorgen, ist offensichtlich.

##### 4.2.2.3. Schlussfolgerung aus den Ergebnissen der ersten Versuchsreihe

Das hier beobachtete Verhalten erinnert stark an die aufgetretenen Fehler während der Probeläufe der Simulationen. In bestimmten Gelenkpositionen kann es sein, dass zwei oder mehr benachbarte Boxen des Agenten miteinander kollidieren. Dies wird allerdings erst dann ein Problem, wenn sie sich verhaken und nicht mehr voneinander lösen können. Die Physik-Engine legt dann zur Durchsetzung der Constraints immer größere Kräfte



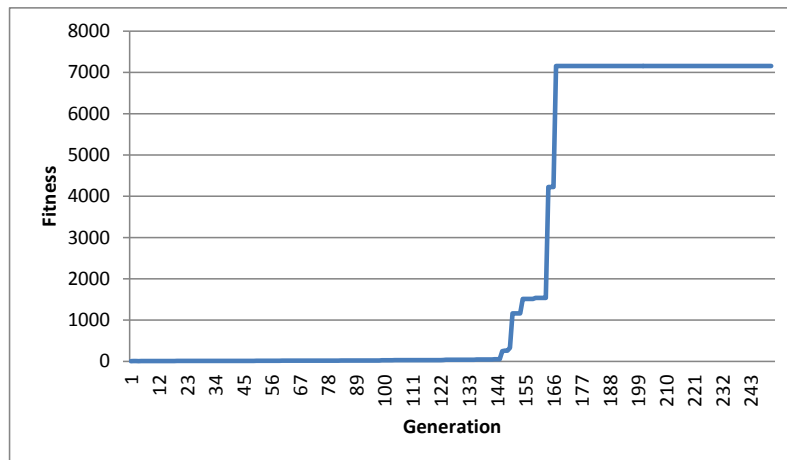


Abbildung 4.4.: Fitnessverlauf von Simulation 3 aus Experiments D1

und Drehmomente an, bis die Boxen schließlich wieder auseinander springen und zurückschlagen. Dieser Ruck kann wiederum den ganzen Agenten erfassen, wodurch er mit enormer Wucht in die Luft geschleudert wird und so eine weite Strecke zurücklegt, ohne den Boden zu berühren.

Offensichtlich war es durch die Evolution möglich, dass sich Agenten herausbilden, deren Körperbau und Künstliches Neuronales Netz exakt diese Lücke in der Simulation ausnutzen können. Es müssen sowohl die Anordnung der Boxen untereinander als auch die Ansteuerung der Gelenke genau zueinander passen, um den beschriebenen Fehler zu provozieren. In den vorherigen Probeläufen fiel das fehlerhafte Verhalten nicht auf, da weder die Länge noch der Umfang der Probeläufe groß genug war, um solche Agenten hervorzubringen.

Das beschriebene Phänomen war auch in zahlreichen weiteren Agenten zu beobachten, aufgezeichnete Pfade hierzu befinden sich im Anhang und auf der beiliegenden DVD.

### 4.2.3. Zweite Versuchsreihe mit modifizierter Fitnessfunktion<sup>12</sup>

Nach einer Auswertung der ersten Versuchsreihe musste festgestellt werden, dass die fehlerhaften Agenten die Versuchsergebnisse stark beeinflussen. Es ließen sich daher keine Schlüsse ziehen, ob die Populationsgröße oder die Verwendung zusätzlicher Sensoren in einem Zusammenhang mit der erreichten Fitness stehen.

Aus diesem Grund wurde eine zweite Versuchsreihe mit demselben Parametersatz wie im ersten Experiment gestartet. Bei dieser Versuchsreihe wurde allerdings die Fitnessfunktion modifiziert, um fehlerhafte Agenten erkennen zu können. In der ersten Versuchsreihe wurde dem Agenten stets die zurückgelegte Entfernung vom Startpunkt als Fitnesswert zugewiesen. Diese äußerst simple Fitnessfunktion gestattete dadurch auch bei fehlerhaften Agenten eine enorme Fitness.

Um Agenten mit irregulärem Verhalten zu bestrafen, wurde den Agenten, welche eine Strecke von mehr als 250 Metern zurücklegten, eine Fitness von 0 zugewiesen. Diese Maßnahme hatte das Ziel, ausschließlich Agenten mit normalem Verhalten zu belohnen, jedoch nicht die mit gewöhnlichen Mitteln erreichbare Fitness künstlich zu beschränken. In der vorherigen Versuchsreihe traten keine Agenten mit kontrollierter Fortbewegung auf, die eine Fitness von 200 oder mehr erreichen. Aus diesem Grund erschien diese Grenze angemessen.

---

<sup>12</sup>Die Versuchsreihe trägt in der Arbeit den Index 2. Die auf der DVD enthaltenen Daten tragen den Index 3, da der vorherige Lauf wegen eines Programmfehlers verworfen werden musste. Die Erörterungen beziehen sich aber auf die Daten mit dem Index 3.

#### 4.2.3.1. Ergebnisse der zweiten Versuchsreihe mit modifizierter Fitnessfunktion

Die in der Tabelle 4.5 abgedruckten Fitnesswerte sind wieder die berechneten Durchschnitte der einzelnen Versuchsläufe eines jeden Experiments.

Tabelle 4.5.: Zweite Versuchsreihe: Maximal erreichte Fitness mit modifizierter Fitnessfunktion

<b>Name des Experiments</b>	<b>Maximal erreichte Fitness (Durchschnitt über je zehn Simulationsläufe)</b>
A2	72
B2	93
C2	159
D2	82
E2	117
F2	134
G2	38
H2	78
I2	109
J2	126
K2	203
L2	176
M2	148
N2	98
O2	124
P2	106
Q2	152
R2	115
S2	60

T2	79
U2	74
V2	89
W2	150
X2	154
Durchschnitt	114
Varianz	1607

#### 4.2.3.2. Auswertung der Ergebnisse aus der zweiten Versuchsreihe

Im Gegensatz zu den in der ersten Versuchsreihe erzielten Ergebnissen sind in der zweiten Versuchsreihe keine extremen Abweichungen in den Fitnesswerten zu erkennen. Die Varianz fällt im Vergleich viel kleiner aus. Dies ist der modifizierten Fitnessfunktion zu verdanken, die unrechtmäßig zurückgelegte Entfernungen von mehr als 250 Metern mit einer Fitness von 0 bestraft.

Da die Modifikation die Fehler der ersten Versuchsreihe vermieden hat, ist nun eine Interpretation der Ergebnisse im Hinblick auf die variierten Parameter möglich.

**Variation der Populationsgröße** Es wurde vermutet, dass eine größere Population auch eine höhere erreichte Fitness des besten Agenten zur Folge hat. Um diese Vermutung zu untersuchen, wurde die durchschnittliche Fitness über alle Experimente mit gleicher Populationsgröße bestimmt.

Tabelle 4.6 zeigt einen Zusammenhang zwischen Populationsgröße und schlussendlich erreichter Fitness des besten Agenten auf. Je größer die Population, desto höher ist durchschnittlich die maximal erreichte Fitness in einem Simulationslauf. Eine Verdreifachung der Populationsgröße führt jedoch nicht zu einer Verdreifachung der erreichten Fitness. Während die Populationsgröße um den Faktor 3 größer wurde, wuchs die erreichte Fitness nur um den Faktor 1,8.

Aus den erhaltenen Daten kann geschlossen werden, dass eine größere

Population zu einer höheren Fitness führt. Dies wird damit begründet, dass die Suche im Suchraum verfeinert wird. Es werden mehr Mutationen und Evaluationen durchgeführt als bei kleineren Populationsgrößen. Insgesamt werden mehr Individuen entwickelt und auf ihre Tauglichkeit geprüft. Damit steigt natürlich auch die benötigte Rechenzeit. Aus den Werten ist weiter ein abnehmender Grenznutzen erkennbar. Eine Vervielfachung der Populationsgröße führt nicht zu einer proportionalen Verbesserung der Fitness.

**Vergleich eines Feed-Forward-Netzes mit einem rekurrenten Netz** Wie in Abschnitt 2.3.3.2 angemerkt, sind rekurrente Künstliche Neuronale Netze turing-vollständig. Sie sind damit theoretisch in der Lage, komplexere Steuerungsaufgaben zu übernehmen als reine Feed-Forward-Netze, die nicht turing-vollständig sind. Um zu untersuchen, ob die Art des verwendeten Künstlichen Neuronalen Netzes einen Einfluss auf die Fitness hat, wurde dieser Parameter variiert. In Tabelle 4.7 findet sich der Vergleich der Fitnesswerte von Agenten mit einem Feed-Forward-Netz im Vergleich zu Agenten mit einem rekurrentem Netz.

Obwohl ein rekurrentes Netz theoretisch komplexere Berechnungen durchführen kann als ein Feed-Forward-Netz, sind keine Vorteile für Agenten mit einem rekurrenten Künstlichen Neuronalen Netz zu erkennen. Tatsächlich ist die erreichte Fitness sogar geringfügig kleiner als bei Agenten mit Feed-Forward-Netz.

Dafür gibt es mehrere mögliche Erklärungen. Ein rekurrentes Künstliches Neuronales Netz erlaubt im Gegensatz zu einem Feed-Forward-Netz auch rückwärts gerichtete Synapsen bzw. Synapsen, die ein Neuron mit sich selbst verbinden. Somit gibt es bei Mutationen, welche das Künstliche Neuronale Netz betreffen, deutlich mehr Möglichkeiten, Synapsen zu erzeugen. Wegen der beschränkten Zahl an Generationen ist es damit möglich, dass durch Mutationen ausschließlich untaugliche Verbindungen in rekurrenten Netzen erzeugt werden, so dass diese ihre Steuerungsaufgabe gar nicht erfüllen können. Bei Feed-Forward-Netzen ist die Zahl an Möglichkeiten geringer, womit die Wahrscheinlichkeit im Vergleich größer ist, die

richtigen Verbindungen mit den richtigen Gewichten herzustellen.

Des Weiteren ist es möglich, dass ein rekurrentes Netz überhaupt nicht benötigt wird. Die Aufgabe, ein Wesen in einer physikalischen dreidimensionalen Welt zu steuern ist nicht trivial. Es ist aber durchaus vorstellbar, dass die Mächtigkeit eines Feed-Forward-Netzes ausreicht, um ein Wesen korrekt zu steuern. Erschwerend kommt hinzu, dass das Künstliche Neuronale Netz die Winkel der Gelenke als Sensorwerte auslesen kann und damit eine Rückkopplung der Ausgabewerte nicht zwangsläufig nötig ist.

Sofern die größere Mächtigkeit rekurrenter Netze nicht benötigt wird, weisen rekurrente Netze nicht zwangsläufig einen Vorteil gegenüber reinen Feed-Forward-Netzen auf. Vielmehr kann es sogar sein, dass durch die größere Anzahl an möglichen Mutationen die Suche nach einer geeigneten Konfiguration erschwert wird.

**Zuschalten eines Lagesensors** Die erstellten Klassen ermöglichen es, einen Lagesensor, welcher die normierte Rotation des Kerns eines Agenten angibt, hinzuzuschalten. Der Sensor bestimmt die Lage des Agenten im Raum, normiert diese und gibt sie als Vektor an das Gehirn des Agenten weiter. Die einzelnen Sensorwerte werden dann wiederum an das Künstliche Neuronale Netz weitergegeben, in welchem sie verwendet werden können.

Ziel der Zuschaltung dieses Sensors war es, herauszufinden, ob die Kenntnis der Lage im Raum den Agenten dazu befähigt, sich gezielter fortzubewegen und seinen Kurs zu halten. Wäre dies der Fall, so müssten Agenten, welche über einen Sensor verfügen, eine höhere Fitness erreichen.

In Tabelle 4.8 ist der Vergleich der durchschnittlichen Fitness von Agenten mit und Agenten ohne Lagesensor abgedruckt.

Agenten, welche über den Lagesensor verfügen, zeigen eine sehr leicht erhöhte durchschnittliche Fitness. Auch wenn zu vermuten ist, dass der Lagesensor theoretisch eine verbesserte Leistung des Agenten ermöglicht, so kann dies nicht nachgewiesen werden.

Das Hinzufügen des Lagesensors führt dazu, dass das Künstliche Neuronale Netz über drei weitere Eingabeneuronen verfügt. Diese Eingabeneuronen erhalten die normierten Komponenten des Sensorvektors als Eingabe-

werte. Zu Beginn sind die Eingabeneuronen jedoch noch nicht mit weiteren (Ausgabe-)Neuronen verbunden. Der Sensor bleibt ohne die richtigen Mutationen ungenutzt.

Durch die Erhöhung der Eingabeneuronenzahl steigt auch gleichzeitig die Anzahl an potentiell mutierbaren Synapsen an. Verschärft tritt diese Situation sogar noch bei rekurrenten Netzen auf. Die Wahrscheinlichkeit, die richtigen Synapsen mit passenden Gewichten auszubilden sinkt mit steigender Zahl an Eingabeneuronen. Es muss darum in Betracht gezogen werden, dass der Vorteil der Sensoren nicht innerhalb der vorgegebenen Zahl an Generationen ausgeschöpft werden kann.

Ein Agent, welcher ohne Lagesensor bereits eine hohe Fitness besitzt, könnte möglicherweise durch das nachträgliche Hinzufügen eines solchen bessere Ergebnisse erzielen. Das Künstliche Neuronale Netz wäre bereits funktionsfähig und auf den Körper des Agenten angepasst. Durch das Hinzufügen des Sensors könnte der Agent sich die weiteren Informationen dann besser zu Nutze machen. Einem zufällig erzeugten Agenten mit noch nicht funktionierendem Künstlichen Neuronalen Netz scheint der Sensor jedoch nicht zu besseren Fitnesswerten zu verhelfen.

**Zuschalten eines Bewegungssensors** Neben dem Lagesensor kann auch ein Bewegungssensor zugeschaltet werden. Dieser misst die Veränderung der Körperposition vom vorherigen zum aktuellen Zeitpunkt. Der gemessene Vektor wird anschließend normiert und dem Gehirn zur Verfügung gestellt. Dieses gibt die Werte wiederum an das Künstliche Neuronale Netz weiter.

Ähnlich wie beim Lagesensor wurde auch beim Bewegungssensor vermutet, dass das Wissen über die aktuelle Bewegungsrichtung den Agenten befähigt, seinen Kurs zu halten und sich geradlinig zu bewegen. Tabelle 4.9 zeigt die Fitnesswerte der Agenten mit und der Agenten ohne Bewegungssensor im Vergleich.

Wie schon zuvor beim Zuschalten des Lagesensors lässt sich auch hier keine Aussage treffen. Die Fitness der Agenten mit Bewegungssensor ist minimal niedriger als die Fitness der Agenten, die über keinen solchen

Sensor verfügten.

Dass sich trotz zusätzlich verwendbarer Sensorinformationen keine Verbesserung der Fitness erkennen lässt, ist wie bereits zuvor zu erklären. Die gesteigerte Zahl an Mutationsmöglichkeiten führt dazu, dass die Künstlichen Neuronalen Netze des Agenten sich nicht so anpassen können, dass sich die Vorteile gegenüber den Agenten ohne Bewegungssensor zeigen. Agenten, welche keine zusätzlichen Sensoren haben, weisen nicht so viele Mutationsmöglichkeiten auf wie Agenten, die über Sensoren verfügen. Bei letzteren ist die Zahl an potentiell erstellbaren Synapsen im Netz größer und damit auch die theoretisch mögliche Leistungsfähigkeit. Einfachere Künstliche Neuronale Netze werden hingegen durch die kleinere Zahl an Möglichkeiten gezielter evolviert und ermöglichen so selbst ohne Zusatzinformationen hohe Fitnesswerte.

##### **4.2.3.3. Schlussfolgerung aus den Ergebnissen der zweiten Versuchsreihe**

Im Gegensatz zur ersten Versuchsreihe waren die Ergebnisse der zweiten Versuchsreihe nicht durch fehlerhafte Agenten verfälscht. Damit ist eine Auswertung der verschiedenen Experimente möglich.

Es wurde gezeigt, dass mit steigender Populationsgröße auch eine höhere Fitness erwartet werden darf. Diese Steigerung erfolgt allerdings nicht proportional, so dass die Populationsgrößen nicht zu umfangreich gewählt werden sollten. Durch die Art des Selektionsmechanismus reduziert sich die Diversität in der Population zu Beginn einer Simulation relativ schnell. Es verbleiben ähnliche Individuen, die sich nur geringfügig unterscheiden. Dadurch wird eine Verbesserung einer Art von Individuen forciert, stark davon abweichende Agenten kommen in der Population nicht mehr vor. Je größer eine Population gewählt wird, desto später kommt es zur konzentrierten Untersuchung eines kleinen Bereichs des Suchraums. Gleichwohl wird dieser – wegen der hohen Zahl an Individuen – gegen Ende intensiver untersucht. Das erklärt wiederum die höhere Fitness. Nicht vergessen werden darf außerdem, dass eine größere Population auch mit mehr Eva-



lationen und damit einer größeren benötigten Rechenzeit einhergeht.

Die zu Beginn vermutete Verbesserung der Fitness durch Verwendung von rekurrenten Künstlichen Neuronalen Netzen konnte nicht bestätigt werden. Die stark steigende Zahl an Mutationsmöglichkeiten im Künstlichen Neuronalen Netz führt vermutlich dazu, dass Mutationen, die die Fitness verbessern, mit einer insgesamt geringeren Wahrscheinlichkeit durchgeführt werden. Damit werden die theoretischen Vorteile eines rekurrenten Netzes aufgehoben. Davon abgesehen scheinen auch einfache Feed-Forward-Netze in der Lage zu sein, die Steuerungsaufgabe zufriedenstellend zu übernehmen, so dass rekurrente Netze nicht zwingend nötig sind. Trotz allem kann ihre Verwendung eine Verbesserung der Fitness bedeuten und wirkt sich nicht zwangsläufig negativ aus.

Auch die Verwendung von Sensoren führte nicht zu der erhofften Verbesserung der Fitness. Ähnlich wie bei den rekurrenten Netzen scheint hier die gesteigerte Zahl an Mutationsmöglichkeiten den Nutzen aufzuheben. Die Wahrscheinlichkeit, gewinnbringende Mutationen durchzuführen, sinkt. Durch das Fehlen der Sensoreingänge einfacher aufgebaute Künstliche Neuronale Netze haben so den Vorteil, schneller vorteilhafte Mutationen zu erfahren und so eine höhere Fitness zu ermöglichen.

### 4.3. Beurteilung der Ergebnisse

Insgesamt kann aus den durchgeführten Experimenten geschlossen werden, dass Evolutionäre Algorithmen eine geeignete Methode darstellen, um dreidimensionale Wesen zu entwickeln, die sich selbstständig fortbewegen. Aus Anfangspopulationen von zufälligen Agenten mit niedrigen einstelligen Fitnesswerten haben sich stets Populationen mit deutlich zweistelligen, teilweise sogar dreistelligen maximalen Fitnesswerten entwickelt. Bezogen auf die Realität bedeutet dies, dass die Wesen zum Teil die Geschwindigkeit eines Joggers erreichen. Fehlerhafte Agenten durchbrachen unter Ausnutzung von Eigenarten der Physik-Engine teilweise sogar die Schallmauer.

Die einfachsten Agenten der Simulationen sind lediglich mit Winkelsensoren und einem einfachen Feed-Forward-Netz ausgestattet. Selbst diese In-

dividuen erfüllen die an sie gestellte Aufgabe, sich möglichst weit von ihrem Startpunkt zu entfernen, zufriedenstellend. Komplexere Agenten meistern die Aufgabe nicht unbedingt besser. Möglicherweise sind diese Agenten erst nach deutlich mehr als den 250 hier durchgeführten Generationen in der Lage, ihre einfachen Artgenossen zu überholen.

### 4.4. Anmerkung zur Robustheit der Ergebnisse

Evolution findet immer zu einem gewissen Zeitpunkt in einer bestimmten Umgebung statt. [FM08] Dies spielt insbesondere bei den hier durchgeführten Experimenten eine übergeordnete Rolle.

Wie sich auch echte Lebewesen an ihre Umwelt anpassen, so passten sich auch die Agenten in den Experimenten an ihre simulierte Umwelt an. Diese Anpassung brachte erstaunliche Verhaltensweisen wie das Ausnutzen von Eigenheiten und Fehlern der Physik-Engine hervor. SIMS weist in seinen Pionierarbeiten bereits darauf hin. [Sim94a, Sim94b]

Neben den erwähnten interessanten Effekten gibt es auch eine bedauerliche Seite dieser Anpassung. Sie geht so weit, dass die erhaltenen Ergebnisse nur wenig robust sind. Ein aus einem bestimmten Genom erzeugter Agent erreicht in einer leeren und neu gestarteten Physik-Welt immer denselben Fitnesswert. Insofern ist seine Leistung stets nachvollziehbar und konstant.

Sobald diese Physik-Welt jedoch manipuliert wird, kann es sein, dass sich das Verhalten des Agenten drastisch ändert. Alleine das Hinzufügen einer einzigen Kollisionsform irgendwo in der Physik-Welt kann dazu führen, dass die Fitness des Agenten massiv geändert wird. Die Ursache für dieses Phänomen ist in der Arbeitsweise der Physik-Engine zu suchen.

JBULLET verfügt über mehrere verschiedene Komponenten, die in ihrem Zusammenspiel die Physik-Simulation ermöglichen. Eine dieser Komponenten ist u.a. für das Durchsetzen von Constraints zuständig.<sup>13</sup> Diese Komponente arbeitet – wie die für diese Arbeit selbst geschriebenen Klassen – ebenfalls mit einem Pseudozufallszahlengenerator. Dieser wird verwendet,

---

<sup>13</sup>Dabei handelt es sich um den sogenannten *SequentialImpulseConstraintSolver*.

um die Reihenfolge der Abarbeitung der Constraints und Gelenke zu bestimmen. Da er immer mit demselben Startwert initialisiert wird, erhält auch die verantwortliche Komponente stets dieselben Pseudozufallszahlen.

Wenn nun aber – in vorherigen Läufen nicht vorhandene – neue Kollisionsformen hinzukommen, so müssen für diese auch Berechnungen durchgeführt werden. In diese Berechnungen fließen die erzeugten Pseudozufallszahlen mit ein, so dass die ursprüngliche Reihenfolge nicht erhalten bleiben kann. In manchen Simulationen spielt die Reihenfolge der Abarbeitung keine Rolle. Der Agent erhält eine ähnliche Fitness wie zuvor. In anderen Fällen hingegen hat sich der Agent so sehr spezialisiert, dass er ausschließlich bei einem exakt gleichen Ablauf eine hohe Fitness erzielen kann. Sobald die Abarbeitungsreihenfolge der Simulation verändert wird, erreicht er nur noch eine geringe Fitness.

Bei normalen Simulationen fällt dieser Aspekt kaum ins Gewicht, da die Physik-Welt zurückgesetzt werden kann, bevor ein Agent hinzugefügt wird. Somit sind die Ergebnisse stets dieselben und bleiben nachvollziehbar. Ein Agent der in einer Welt ohne Hindernisse hervorragende Ergebnisse liefert, könnte aber in einer Welt mit Hindernissen komplett versagen. Und das ohne ein einziges Hindernis zu berühren. Dies ist deshalb möglich, weil alleine die Änderung der Welt die Art der Berechnung ändert, an welche sich der Agent ebenfalls angepasst hat.

Das beschriebene Phänomen betrifft leider auch die grafische Ausgabe durch das *VideoPlugin3D*. Dieses stellt, um die Physik-Welt grafisch darstellen zu können, Anfragen an die Physik-Engine. Von dieser werden daraufhin zusätzliche Berechnungen durchgeführt, welche wiederum die Pseudozufallszahlen abfragen. Dadurch wird aber die Abarbeitungsfolge verändert, was wie erklärt eine Veränderung der Physik-Welt darstellt. Eine Beobachtung eines Agenten ist aus diesem Grund mit dem *VideoPlugin3D* häufig nicht möglich, da das Verhalten des Agenten beeinflusst wird. Die Aufzeichnung des Bewegungspaths eines Agenten beeinflusst die Simulation hingegen nicht, so dass diese bedenkenlos durchgeführt werden kann.

Zusammenfassend lässt sich sagen, dass die Ergebnisse der Simulationen zwar exakt nachvollziehbar aber wenig robust sind. Die Spezialisierung der

Agenten durch die Evolution geht so weit, dass sie sich nicht nur an eine sterile, vermeintlich perfekt simulierte Umwelt anpassen. Vielmehr passen sie sich an eine ganz spezielle, eben nicht perfekte Physik-Welt an. Sie passen sich sogar an die Abarbeitungsreihenfolge der Berechnungen an, so dass eine kleine Änderung zu großen Fitnessänderungen führen kann. Wünschenswert wäre hier der Entwurf und die Entwicklung robuster Agenten, die ein Verhalten entwickeln, welches auch in verschiedenen Umgebungen eine hohe Fitness verspricht.

Tabelle 4.6.: Vergleich der durchschnittlich erreichten maximalen Fitnesswerte bei unterschiedlichen Populationsgrößen

Populationsgröße	Namen der Experimente	Maximal erreichte Fitness (Durchschnitt)
50	A2, B2, G2, H2, M2, N2, S2, T2	83
100	C2, D2, I2, J2, O2, P2, U2, V2	109
150	E2, F2, K2, L2, Q2, R2, W2, X2	150

Tabelle 4.7.: Vergleich der durchschnittlich erreichten maximalen Fitnesswerte zwischen Feed-Forward-Netz und rekurrentem Netz

Art des Netzes	Namen der Experimente	Maximal erreichte Fitness (Durchschnitt)
Feed-Forward-Netz	A2, C2, E2, G2, I2, K2, M2, O2, Q2, S2, U2, W2	117
Rekurrentes Netz	B2, D2, F2, H2, J2, L2, N2, P2, R2, T2, V2, X2	111

Tabelle 4.8.: Vergleich der durchschnittlich erreichten maximalen Fitnesswerte zwischen Agenten mit und Agenten ohne Lagesensor

Lagesensor	Namen der Experimente	Maximal erreichte Fitness (Durchschnitt)
nicht verfügbar	A2, B2, C2, D2, E2, F2, M2, N2, O2, P2, Q2, R2	111
verfügbar	G2, H2, I2, J2, K2, L2, S2, T2, U2, V2, W2, X2	117

Tabelle 4.9.: Vergleich der durchschnittlich erreichten maximalen Fitnesswerte zwischen Agenten mit und Agenten ohne Bewegungssensor

Bewegungssensor	Namen der Experimente	Maximal erreichte Fitness (Durchschnitt)
nicht verfügbar	A2, B2, C2, D2, E2, F2, G2, H2, I2, J2, K2, L2	116
verfügbar	M2, N2, O2, P2, Q2, R2, S2, T2, U2, V2, W2, X2	112

## 5. Zusammenfassung und Ausblick

Der Grund und Boden,  
auf dem alle unsere  
Erkenntnisse und  
Wissenschaften ruhen,  
ist das Unerklärliche.

---

*Arthur Schopenhauer*

### 5.1. Zusammenfassung der Arbeit

Die vorliegende Arbeit befasst sich mit der Untersuchung künstlicher dreidimensionaler Wesen, die durch Evolution die Fähigkeit entwickeln, sich selbstständig in einer simulierten Welt fortzubewegen.

Zu Beginn der Arbeit wurden durch eine Literaturrecherche die nötigen Grundlagen erarbeitet. Auf diesen Grundlagen aufbauend wurden Ideen entwickelt, wie die Untersuchung der Wesen systematisch erfolgen könnte. Um die Versuche umsetzen zu können, wurde auf Basis des bereits bestehenden EAS-Frameworks eine Simulationsumgebung entworfen und programmiert. Diese Umgebung sollte physikalisch korrekte Simulationen im dreidimensionalen Raum ermöglichen.

Zur Entwicklung des Simulationswerkzeugs wurde die quelloffene Physik-Engine JBULLET verwendet. Die Engine musste in das EAS-Framework eingebunden werden, um sie für Simulationen nutzbar zu machen. Da die

Verwendung der Engine selbst bei einfachen Simulationen vergleichsweise viele Befehle erfordert, wurden vereinfachende Klassen erstellt. Auf Basis dieser Klassen können weitere Projekte und Versuche programmiert werden, die dann mit geringem Aufwand auf die Physik-Engine zugreifen.

Nachdem die Simulationsumgebung eingerichtet war, konnten die eigentlichen Versuchsobjekte – die Agenten – entworfen und programmiert werden. Ein solcher dreidimensionaler Agent besteht immer aus mindestens zwei Boxen, welche miteinander durch ein Gelenk verbunden sind. Dieses Gelenk ist nur um eine einzige Achse drehbar, alle anderen Achsen bleiben starr. Der Agent kann den Gelenkwinkel mittels ihm zur Verfügung stehender Sensoren messen und über Aktoren einstellen. Die Sensorwerte werden dabei von seinem Gehirn ausgelesen und fließen in Berechnungen mit ein, die die Ausgabewerte erzeugen. Diese Ausgabewerte werden wiederum von dem Gehirn an die Aktoren weitergegeben, die die gewünschte Winkeländerung durchführen. Ein Künstliches Neuronales Netz bildet den Kern des Gehirns eines Agenten.

Die Agenten basieren auf künstlichen Genomen, in welchen sämtliche Informationen gespeichert sind, die erforderlich sind, um einen Agenten zu erzeugen. Diese Informationen umfassen sowohl den physischen Aufbau als auch das Künstliche Neuronale Netz des Agenten. Damit eine Evolution überhaupt erst möglich wird, mussten verschiedene Mutationsoperatoren entwickelt werden. Zu den Operatoren gehören solche, die den Aufbau des Agenten verändern und solche, die sein Künstliches Neuronales Netz beeinflussen. Auf eine Rekombinationsmöglichkeit zweier Agenten wurde verzichtet, da diese eine Reihe von Problemen aufwirft.

Die fertig eingerichtete Simulationsumgebung und die erstellten Agenten machten eine Untersuchung der Evolution schließlich möglich. Dazu wurden Experimente entworfen, in welchen diejenigen Agenten eine hohe Bewertung erhalten, die sich möglichst weit von ihrem Startpunkt entfernen. So wurden viele Populationen von Agenten zufällig erzeugt, simuliert und evaluiert. Die besten Agenten durften ihre Genome replizieren, um in der nächsten Generation wieder in der Population vorhanden zu sein. Außerdem wurden ihre Genome kopiert und mutiert, um leicht veränderte



Nachkommen zu erzeugen.

Der verwendete Evolutionäre Algorithmus brachte künstliche Wesen hervor, die sich selbstständig in einer physikalischen dreidimensionalen Welt bewegen können. Einige der Parameter, welche die Evolution beeinflussen können, wurden näher untersucht. Dabei wurde festgestellt, dass größere Populationen zu gesteigerten Fitnesswerten führen. Es konnte aber nicht bestätigt werden, dass die Verwendung rekurrenter Netze einen Vorteil gegenüber der Verwendung normaler Feed-Forward-Netze bietet. Weiterhin wurde gezeigt, dass die Agenten keine Vorteile durch die zusätzliche Verwendung von Lage- oder Bewegungssensoren erhalten.

## 5.2. Ausblick

### 5.2.1. Weitere Forschungsmöglichkeiten

Auch wenn sich bereits komplexe Aufgaben wie das selbstständige Fortbewegen in einer dreidimensionalen Welt mittels Evolutionärer Algorithmen bewältigen lassen, besteht noch viel Potential für weitere Forschung.

Die in dieser Arbeit entwickelten Agenten weisen ein wenig robustes Verhalten auf, so dass selbst Agenten mit hervorragenden Fitnesswerten in veränderten Umgebungen versagen können. Eine Anwendung in kritischen Bereichen erfordert jedoch ein hohes Maß an Robustheit. Empfindlich auf Veränderungen in der Umwelt reagierende Agenten stellen ein Problem dar, da ihr Verhalten nicht vorhersagbar ist. Die Steigerung der Robustheit der Agenten und die Fähigkeit, auch in veränderten Umgebungen weiterhin konstante Leistungen zu erbringen, wäre daher ein interessantes Forschungsziel.

Es gibt neben Mutationen auch noch die Möglichkeit der Rekombination, um veränderte Nachkommen zu erzeugen. Hier wurde ausschließlich die Mutation betrachtet, da die Rekombination zweier unterschiedlicher Agenten mit Schwierigkeiten verbunden ist. Es entstanden fast ausschließlich Nachkommen, die aufgrund zu großer Unterschiede in den Elternteilen keine hohe Fitness besaßen und schlechter abschnitten als ihre Eltern. Mög-

licherweise gibt es andere Rekombinationsmethoden, die es gestatten, das Erbgut zweier Eltern so zu kombinieren, dass tatsächlich bessere dreidimensionale Agenten entstehen. Die simultane Verwendung von Mutationen und Rekombinationen in einem Evolutionären Algorithmus würde eventuell die Diversität der Population sowie deren Fitness steigern. Daher erscheint auch diese Richtung vielversprechend.

Mit dem entwickelten, relativ einfachen, Genom in *String*-Repräsentation wurde nur eine von vielen verschiedenen Möglichkeiten gewählt. Interessant wäre es, zu überprüfen, wie geeignet verschiedene Genom-Repräsentationen sind. In der Literatur finden sich vor allem graphenorientierte Genome, die zwar einerseits ausdrucksmächtiger, andererseits aber auch deutlich komplexer sind. Die Untersuchung, ob die gesteigerte Komplexität wirklich einen Vorteil bringt, könnte Grundlage für weitere Forschung sein.

In der vorliegenden Arbeit wurden stets gleichzeitig der Körper und das Gehirn eines Wesens evolviert. Dies eröffnet zwar einerseits vielfältige Möglichkeiten für verschiedene Formen und Steuerungen, steigert andererseits aber auch die Komplexität der Suche nach einem geeigneten Kandidaten. Einige Arbeiten befassen sich mit dem Thema der Evolution von Künstlichen Neuronalen Netzen in sich nicht verändernden zweidimensionalen Wesen. Es könnte überprüft werden, ob sich die Ergebnisse auch auf dreidimensionale Wesen übertragen lassen, indem manuell ein Körper erstellt wird. Dieser Körper wird dann nicht mehr evolviert sondern mit verschiedenen Künstlichen Neuronalen Netzen versehen, deren Entwicklung dann untersucht werden kann.

Die Standardsimulationswelt dieser Arbeit besteht aus einem Raum mit flachem Boden, auf welchem sich die Agenten fortbewegen. Denkbar wäre es, eine Simulation von schwimmenden, fliegenden oder springenden Wesen durchzuführen. Dazu müsste die Physik-Engine aber erweitert und angepasst werden, da solche Simulationen in der aktuellen Version nicht direkt bzw. nur über Umwege durchgeführt werden können.

Zuletzt bietet das, um die dreidimensionale Physik-Engine erweiterte, EAS-Framework die Möglichkeit, ganz neue Experimente zu entwerfen und durchzuführen. Im Rahmen der Implementierung der Engine wurden ab-

strakte Klassen programmiert, die die Einstiegshürden für neue dreidimensionale Projekte stark senken sollen. Damit ist es vergleichsweise einfach möglich, eigene Versuchsreihen und Projekte in einer physikalisch korrekt simulierten dreidimensionalen Welt durchzuführen.

### 5.2.2. Anwendung der Ergebnisse

Die Erforschung der Evolution von dreidimensionalen Wesen und Maschinen in einer simulierten Welt steht erst am Anfang. Mit steigender verfügbarer Rechenleistung und immer neuen Methoden wird es möglich sein, das hier verwendete Konzept auch in großem Maßstab einzusetzen, um komplexeste Konstrukte zu evolvieren. Bereits heute werden Evolutionäre Algorithmen und Künstliche Neuronale Netze erfolgreich zur Lösung von Problemen und für Steuerungsaufgaben eingesetzt.

Die in fast allen Bereichen steigende Komplexität von Konstruktions- und Steuerungsaufgaben macht es dem Menschen immer schwerer, Schritt zu halten. Ein Nachahmen der Evolution könnte dazu beitragen, dass die entstehenden Probleme gelöst werden können, ohne dass ein genaues Verständnis der einzelnen Zusammenhänge im Detail erforderlich ist. Die Konstruktion erfolgt dann nicht mehr durch Ingenieure sondern durch evolutionäre Vorgänge. Sobald es dann gelingt, die – aus den Simulationen hervorgegangen – Ergebnisse auch in der Realität umzusetzen, könnten Konstrukte erzeugt werden, die die Komplexität aller heute vorhandenen Maschinen bei weitem übertreffen.



## **A. Vorstellung einiger Simulationsläufe**

An dieser Stelle sollen einige typische und interessante Simulationsläufe der Versuchsreihen vorgestellt werden. Die Daten und Abbildungen zu allen Simulationsläufen finden sich auf der beiliegenden DVD.

### **A.1. Erste Versuchsreihe mit normaler Fitnessfunktion**

#### **A.1.1. Verlauf der Fitness**

In den Abbildungen A.1, A.2 und A.3 ist gut der charakteristische Verlauf der Fitness aus Experimenten der ersten Versuchsreihe zu erkennen. Zunächst ist keine auffällige Erhöhung der Fitness zu beobachten. Nach mehreren Generationen tritt dann eine sprunghafte Steigerung der Fitness um hunderte oder gar tausende Punkte auf. Die normale Entwicklung von Agenten spielt quasi keine Rolle mehr, da sie von der Ausnutzung von Eigenheiten der Physik-Engine überschattet wird.

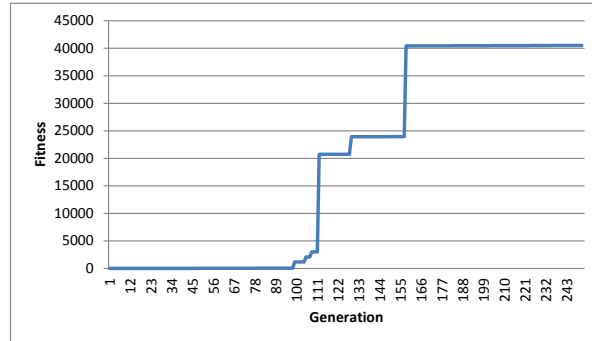


Abbildung A.1.: Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment *L1*

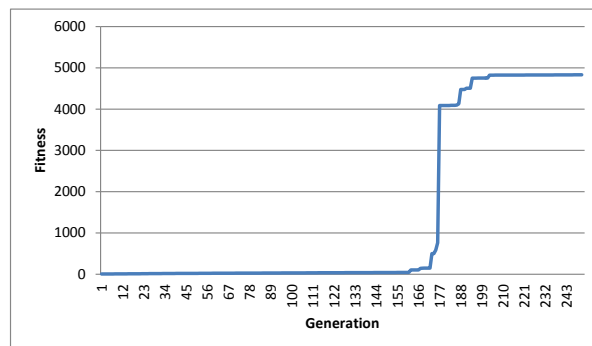


Abbildung A.2.: Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment *U1*

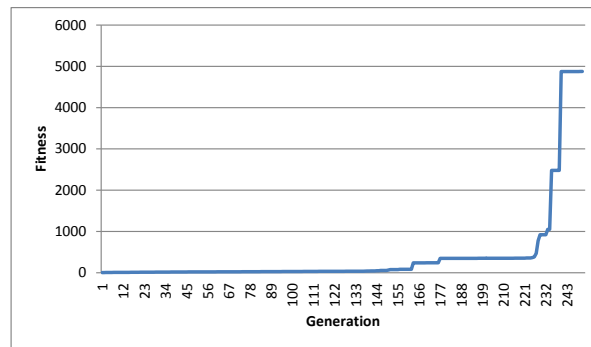


Abbildung A.3.: Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment *V1*

### A.1.2. Pfade

Die Abbildungen A.4, A.5 und A.6 zeigen jeweils den zurückgelegten Weg eines fehlerhaften Agenten im Vergleich zum Pfad eines normalen Agenten. Auffällig sind die enormen Unterschiede in den Fitnesswerten, die für die hohe Varianz der Experimente sorgen. Während der normale Agent aus Abbildung A.5 einen relativ geradlinigen Weg zurückgelegt hat, bewegten sich die anderen normalen Agenten in einem Kreisbogen.

In Abbildung A.7 ist sogar eine stark gezackte Kurve zu erkennen. Der Agent hat tatsächlich einen sehr viel größeren Weg zurückgelegt, als die Fitness zunächst vermuten lässt. Es wird jedoch nur die direkte Verbindungsstrecke von Anfangs- und Endpunkt in die Bewertung mit einbezogen.

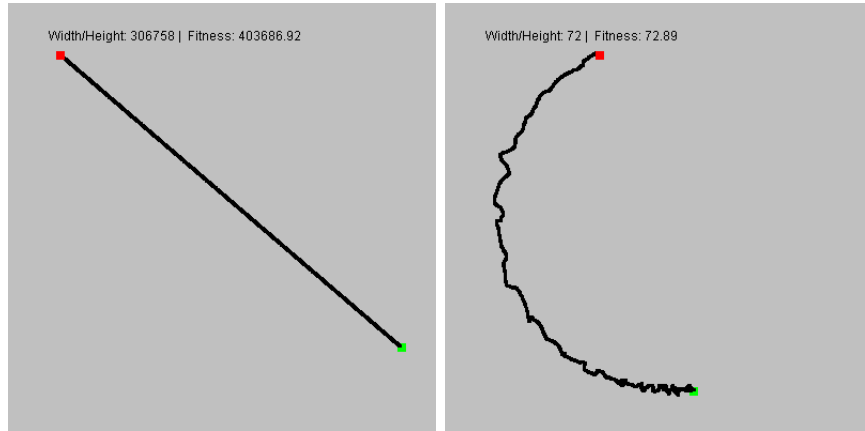


Abbildung A.4.: Zurückgelegte Pfade eines fehlerhaften Agenten (links) und eines normalen Agenten (rechts) des Experiments *L1*. Der Startpunkt ist grün markiert, der Endpunkt rot.

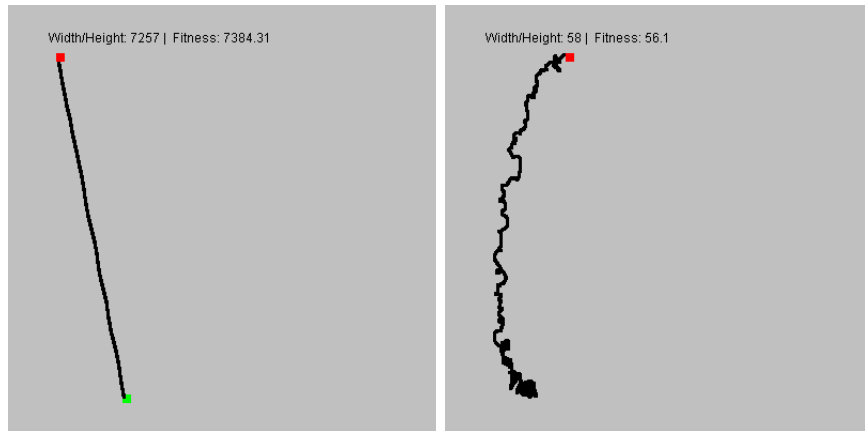


Abbildung A.5.: Zurückgelegte Pfade eines fehlerhaften Agenten (links) und eines normalen Agenten (rechts) des Experiments *U1*. Der Startpunkt ist grün markiert, der Endpunkt rot.



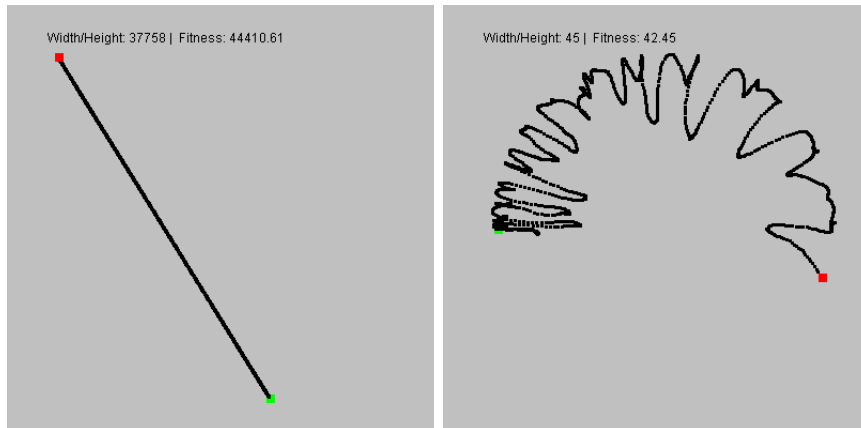


Abbildung A.6.: Zurückgelegte Pfade eines fehlerhaften Agenten (links) und eines normalen Agenten (rechts) des Experiments V1. Der Startpunkt ist grün markiert, der Endpunkt rot.

## A.2. Zweite Versuchsreihe mit modifizierter Fitnessfunktion

### A.2.1. Verlauf der Fitness

Das Ausnutzen von Eigenheiten der Physik-Engine führte in der ersten Versuchsreihe zu sehr hohen Varianzen. Die Fitnessfunktion wurde angepasst, um die Entwicklung fehlerhafter Agenten zu unterbinden. In den Abbildungen A.7, A.8 und A.9 sind typische Fitnessverläufe für Experimente der zweiten Versuchsreihe dargestellt. Im Gegensatz zu den fehlerbehafteten Experimenten aus der ersten Versuchsreihe steigt die Fitness hier in vielen kleinen Schritten an. Sie bleibt außerdem immer im niedrigen dreistelligen oder gar nur im zweistelligen Bereich.

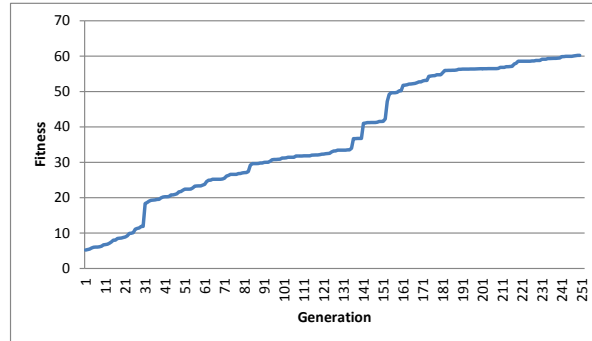


Abbildung A.7.: Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment  $S2$

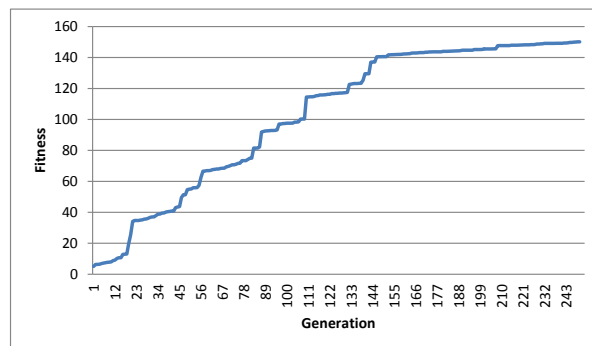


Abbildung A.8.: Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment  $W2$

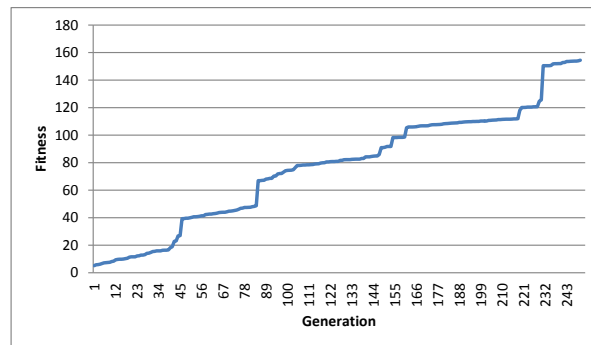


Abbildung A.9.: Verlauf der durchschnittlich erreichten maximalen Fitness der Simulationsläufe in Experiment *X2*

### A.2.2. Pfade

Die Experimente der zweiten Versuchsreihe brachten eine Vielzahl an interessanten Individuen hervor. In Abbildung A.10 sind zwei Pfade von Agenten des Experiments *S2* zu sehen. Der erste der beiden Agenten legt einen stark verwinkelten Weg zurück und erreicht eine durchschnittliche Fitness.

Der zweite Agent aus A.10 zeigt ein bemerkenswertes Verhalten. Die modifizierte Fitnessfunktion bestrafte Agenten, welche eine Entfernung von 250 Metern oder mehr zurücklegten. Der Pfad des Agenten zeigt ein genau darauf angepasstes Verhalten. Der Agent bewegt sich in der Nähe des Startpunkts (grünes Quadrat) hin und her, um schließlich eine Strecke von etwas über 249 Metern zurückzulegen. Er nutzt so den zulässigen Bereich fast komplett aus. Hätte er sich einen Meter weiter bewegt, wäre ihm eine Fitness von null zugewiesen worden. Offensichtlich hat sich der Agent exakt an die Umgebung, die vorgegebene Simulationszeit angepasst und so annähernd die maximal mögliche Fitness erreicht. Dies gelang ihm unter kontrollierter Ausnutzung einer Eigenheit der Physik-Engine.

In A.11 und A.12 sind die Pfade von vier verschiedenen Agenten aus den

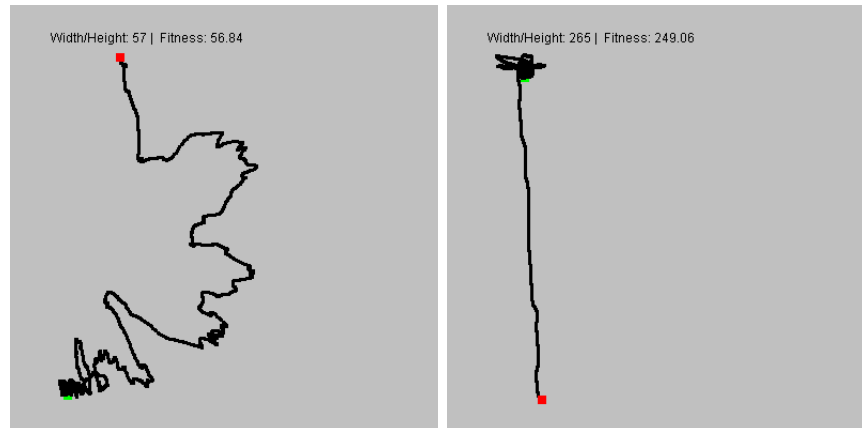


Abbildung A.10.: Zurückgelegte Pfade von zwei ausgewählten Agenten des Experiments  $S2$ . Der Startpunkt ist grün markiert, der Endpunkt rot.

Experimenten  $W2$  und  $X2$  abgebildet. Jeder der Agenten hat eine eigene Art, sich fortzubewegen, und damit einen unterschiedlichen Pfadverlauf entwickelt. Neben relativ geradlinigen Verläufen sind auch ineffiziente Bewegungen zu sehen. Die rechten Pfade von A.11 und A.12 zeigen beispielsweise, dass die zurückgelegte Strecke oft deutlich länger sein kann, als die Fitness es vermuten lässt, wenn der Agent sich nicht geradlinig sondern in vielen kleinen Kurven und Kreisen bewegt.

## A.2. ZWEITE VERSUCHSREIHE MIT MODIFIZIERTER FITNESSFUNKTION

---



Abbildung A.11.: Zurückgelegte Pfade von zwei ausgewählten Agenten des Experiments *W2*. Der Startpunkt ist grün markiert, der Endpunkt rot.

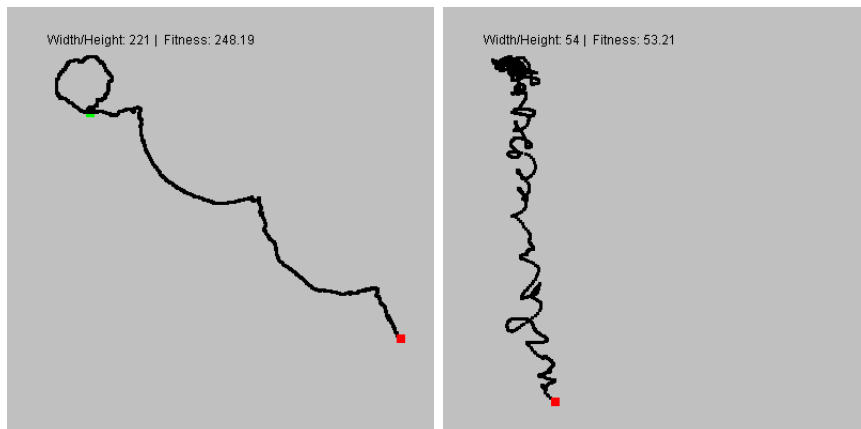


Abbildung A.12.: Zurückgelegte Pfade von zwei ausgewählten Agenten des Experiments *X2*. Der Startpunkt ist grün markiert, der Endpunkt rot.



## B. DVD

Auf der beiliegenden DVD finden sich folgende Dateien.

- Diese Arbeit als PDF.
- Die verwendeten Quellen soweit vorhanden als PDF.
- Alle verwendeten Abbildungen.
- Das zum Abgabezeitpunkt aktuelle EAS-Framework mit allen für die Experimente nötigen Klassen.
- Die verwendeten JBULLET-Klassen inklusive Quelltext.
- Die JOSCHKA-Dateien der durchgeführten Versuchsreihen.
- Die Ergebnisse und Auswertungen der Simulationsläufe.





# Literaturverzeichnis

- [Bon07] BONN, Matthias: *JoSchKa: Job Scheduling Karlsruhe*. Website abgerufen am 30.09.2011. <http://people.aifb.kit.edu/mbo/joschka/>. Version: November 2007
- [Bon08] BONN, Matthias: *JoSchKa: Jobverteilung in heterogenen und unzuverlässigen Umgebungen*, Fakultät für Wirtschaftswissenschaften (Fak. f. Wirtschaftswiss.) Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB) Karlsruher Institut für Technologie (KIT), Diss., 2008
- [Bul11] BULLETPHYSICS.ORG: *Bullet Physics Wiki*. Forum. [http://www.bulletphysics.org/mediawiki-1.5.8/index.php/Main\\_Page](http://www.bulletphysics.org/mediawiki-1.5.8/index.php/Main_Page). Version: 2011
- [Col11] COLLING, Dominik: *Evolution neuronaler Netze in einer physikalischen 2D-Umgebung*. Bachelorarbeit, September 2011
- [Cou10] COUMANS, Erwin: *Bullet 2.76 Physics SDK Manual [PDF]*, 2010. [http://bulletphysics.com/ftp/pub/test/physics/Bullet\\_User\\_Manual.pdf](http://bulletphysics.com/ftp/pub/test/physics/Bullet_User_Manual.pdf). – [Stand 30.09.2011]
- [CR06] CAMPBELL, Neil A. ; REECE, Jane B. ; MARKL, Jürgen (Hrsg.): *Biologie*. 6. Aufl., geringfügig überarb. Nachdr. München [u.a.] : Pearson Studium, 2006. – ISBN 3-8273-7180-5 ; 978-3-8273-7180-5

- [CS97] CHURCHLAND, Patricia S. ; SEJNOWSKI, Terrence J.: *Grundlagen zur Neuroinformatik und Neurobiologie – The computational brain in deutscher Sprache*. Braunschweig : Vieweg, 1997 (Computational intelligence). – ISBN 3-528-05428-X
- [Dar59] DARWIN, Charles R.: *On the Origin of Species by Means of Natural Selection*. London, 1859
- [Dow04] DOWNING, Keith L.: Development and the Baldwin effect. In: *Artificial Life* 10 (2004), Nr. 1, S. 39–63
- [Dvo10a] DVORAK, Martin: *JBullet - Java port of Bullet Physics Library*. Website abgerufen am 30.09.2011. <http://jbullet.advel.cz/>. Version: 2010
- [Dvo10b] DVORAK, Martin: *JBullet - Java port of Bullet Physics Library [JavaDoc Documentation]*, 2010. <http://jbullet.advel.cz/javadoc/>. – Dokumentation abgerufen am 30.09.2011
- [Egg97] EGGENBERGER, Peter: Evolving morphologies of simulated 3D organisms based on differential gene expression. In: *Proceedings of the Fourth European Conference on Artificial Life* MIT Press Cambridge, MA, 1997, S. 205–213
- [Fah91] FAHLMAN, Scott E.: The recurrent cascade-correlation architecture. In: *Advances in neural information processing systems* 3 (1991), May, S. 190–196
- [FL90] FAHLMAN, Scott E. ; LEBIERE, Christian: The cascade-correlation learning architecture. In: *Advances in neural information processing systems* 2 (1990), Nr. 2, S. 524–532
- [FM08] FLOREANO, Dario ; MATTIUSI, Claudio: *Bio-inspired artificial intelligence: theories, methods, and technologies*. Cambridge, Mass. [u.a.] : MIT Press, 2008 (Intelligent robotics and autonomous agents). – ISBN 978-0-262-06271-8

- [Fut07] FUTUYMA, Douglas J.: *Evolution: Das Original mit Übersetzungshilfen*. 1. Aufl. Heidelberg : Elsevier, Spektrum Akademischer Verlag, 2007
- [GKK04] GERDES, Ingrid ; KLAWONN, Frank ; KRUSE, Rudolf: *Evolutionäre Algorithmen: Genetische Algorithmen, Strategien und Optimierungsverfahren, Beispielanwendungen*. 1. Aufl. Wiesbaden : Vieweg, 2004 (Computational intelligence). – ISBN 3-528-05570-7
- [Hak96] HAKEN, Hermann: *Principles of brain functioning: A synergetic approach to brain activity, behavior and cognition*. Berlin : Springer, 1996 (Springer series in synergetics ; 67). – ISBN 3-540-58967-8
- [Hel08] HELAOUI, Rim: *Morphological Development of Artificial Embodied Organisms under the Control of Gene Regulatory Networks*. Diplomarbeit, November 2008
- [Hyö96] HYÖTYNIEMI, Heikki: Turing machines are recurrent neural networks. In: *Proc. of STeP'96-Genes, Nets and Symbols* (1996), 13-24. <http://lipas.uwasa.fi/stes/step96/step96/hyotyniemi1/>
- [KB03] KUMAR, Sanjeev ; BENTLEY, Peter J.: Biologically inspired evolutionary development. In: *Proceedings of the 5th international conference on Evolvable systems: From biology to hardware* Springer-Verlag, 2003, S. 57-68
- [KP11] KÖNIG, Lukas ; PATHMAPERUMA, Daniel: *EAS-Framework Website*. Website abgerufen am 30.09.2011. <http://sourceforge.net/projects/eas-framework/>. Version: 2011
- [Krč07] KRČAĤ, Peter: Evolving virtual creatures revisited. In: *GECCO* Bd. 7, 2007, S. 341

- [Krč08] KRČAĤ, Peter: Towards efficient evolution of morphology and control. In: *Proceedings of the 10th annual conference on Genetic and evolutionary computation* ACM, 2008, S. 287–288
- [Krč10] KRČAĤ, Peter: Solving deceptive tasks in robot body-brain co-evolution by searching for behavioral novelty. In: *10th International Conference on Intelligent Systems Design and Applications*. Cairo, Egypt, 2010, S. 248–289
- [Krč11] KRČAĤ, Peter: *ERO*. Website abgerufen am 30.09.2011. <http://ero.matfyz.cz/>. Version: 2011
- [KT10] KRČAĤ, Peter ; TOROPILA, Daniel: Combination of Novelty Search and Fitness-Based Search Applied to Robot Body-Brain Co-Evolution. In: ITOH, T. (Hrsg.) ; SUZUKI, K. (Hrsg.): *T. Itoh and K. Suzuki (Ed.), Proceedings of the 13th Czech-Japan Seminar on Data Analysis and Decision Making in Service Science*. Otaru, Japan, 2010, S. 25–30
- [Lam09] LAMARCK, Jean-Baptiste: *Philosophie Zoologique*. C. Martins, 1809
- [LLD07] LASSABE, N. ; LUGA, H. ; DUTHEN, Y.: A new step for artificial creatures. In: *Artificial Life, 2007. ALIFE'07. IEEE Symposium on IEEE*, 2007, S. 243–250
- [LS11] LEHMAN, Joel ; STANLEY, Kenneth O.: Evolving a Diversity of Creatures through Novelty Search and Local Competition. In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)* (2011)
- [Mü10] MÜLLER, Benedikt: *Neuroevolution in Roboterschwärmen*. Studienarbeit, Juli 2010
- [Mic07] MICONI, Thomas: *The Road to Everywhere: Evolution, Complexity and Progress in Natural and Artificial Systems*, University of Birmingham, Diss., 2007

- [MK98] MARTINEZ, Joe L. ; KESNER, Raymond P. ; MARTINEZ, Joe L. (Hrsg.): *Neurobiology of learning and memory*. San Diego [u.a.] : Academic Press, 1998. – ISBN 0–12–475655–7
- [MP43] MCCULLOCH, Warren ; PITTS, Walter: A logical calculus of the ideas immanent in nervous activity. In: *Bulletin of Mathematical Biology* Bd. 5, 1943, S. 115–133
- [Nag10] NAGEL, Christian: *Evolution von Kooperation in Roboterschwärmen*. Bachelorarbeit, September 2010
- [NF01] NOLFI, Stefano ; FLOREANO, Dario: *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. 2. print. Cambridge, Mass. : MIT Pr., 2001 (Intelligent robots and autonomous agents – A Bradford book). – ISBN 0–262–64056–2
- [Ope11] OPEN SOURCE INITIATIVE: *The zlib/libpng License*. Website abgerufen am 30.09.2011. <http://opensource.org/licenses/zlib-license.php>. Version: 2011
- [Pat08] PATHMAPERUMA, Daniel: *Lernende und selbstorganisierende Putzroboter*. Diplomarbeit, April 2008
- [Pri11] PRICE, David J.: *Building brains: An introduction to neural development*. 1. publ., 1. impr. Oxford [u.a.] : Wiley-Blackwell, 2011. – ISBN 978–0–470–71230–6
- [RMS91] RITTER, Helge ; MARTINETZ, Thomas ; SCHULTEN, Klaus ; WAHLSTER, Prof. Dr. W. (Hrsg.): *Neuronale Netze: Eine Einführung in die Neuroinformatik selbstorganisierender Netzwerke*. 2., erw. Aufl. Bonn [u.a.] : Addison-Wesley, 1991. – ISBN 3–89319–172–0 ; 3–89319–131–3
- [Ros58] ROSENBLATT, F.: The perceptron: A probabilistic model for information storage and organization in the brain. In: *Psychological Review* Bd. 65, American Psychological Association, 1958, S. 386–408

- [Sim94a] SIMS, Karl: Evolving 3D morphology and behavior by competition. In: *Artificial Life* 1 (1994), Nr. 4, S. 353–372
- [Sim94b] SIMS, Karl: Evolving virtual creatures. In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* ACM, 1994, S. 15–22
- [SS96] SMITH, John M. ; SZATHMÁRY, Eörs: *Evolution: Prozesse, Mechanismen, Modelle*. Heidelberg : Spektrum, Akad. Verl., 1996. – ISBN 3–8274–0022–8
- [Sta06] STANLEY, Kenneth O.: Comparing artificial phenotypes with natural biological patterns. In: *Proceedings of the genetic and evolutionary computation conference (GECCO) workshop program, New York, NY*, 2006
- [Sta11] STANLEY, Kenneth O.: *The NeuroEvolution of Augmenting Topologies (NEAT) Users Page*. Website abgerufen am 30.09.2011. <http://www.cs.ucf.edu/~kstanley/neat.html>. Version: 2011
- [Wei07] WEICKER, Karsten: *Evolutionäre Algorithmen*. 2., überarb. und erw. Aufl. Wiesbaden : Teubner, 2007 (Leitfäden der Informatik, Lehrbuch Informatik). – ISBN 978–3–8351–0219–4